

Содержание

Введение	3
Глава 1	
Основные понятия объектно-ориентированного программирования	5
1.1. Описание класса	6
1.2. Создание и использование объектов	9
1.3. Конструкторы и деструкторы	11
Глава 2	
Технология разработки приложения Windows в среде Visual Studio 2008 на языке C++/CLI	17
2.1. Создание проекта	17
2.2. Окно сведений об объекте	28
2.3. Управление окнами документов	31
2.4. Редактор кода и режим проектирования	31
2.5. Конструктор форм	32
2.6. Использование нескольких форм	33
2.7. Свойства формы	37
Глава 3	
Исследование базовых компонентов интерфейса приложения	41
3.1. Компоненты Button, Panel, Label, TextBox, PictureBox	41
3.2. Использование переменных различных типов	56
3.3. Разработка приложений с использованием классов	61
Глава 4	
Наследование классов	71
Глава 5	
Полиморфизм	91
5.1. Перегрузка функций	91
5.2. Перегрузка операций	94
5.3. Виртуальные функции	99
5.4. Шаблоны	107
Глава 6	
Использование расширенных элементов управления	115
6.1. Компоненты список (ListBox, ComboBox), флажок (CheckBox) и переключатель (RadioButton)	115

Глава 7	
Компоненты, обеспечивающие диалог с пользователем	127
7.1. Компоненты SplitContainer, MenuStrip, ContextMenuStrip, RichTextBox, OpenFileDialog, SaveFileDialog, ColorDialog, FontDialog.....	127
Глава 8	
Обеспечение работы с файлами и Web-страницами	145
8.1. Программирование операций файлового ввода-вывода в Windows-приложениях	145
8.2. Загрузка текстовых строк из файла в компонент ComboBox	147
8.3. Компонент WebBrowser.....	152
Глава 9	
Технология разработки многостраничных приложений.....	161
9.1. Компонент TabControl	161
9.2. Компоненты DateTimePicker, Timer, ProgressBar	163
Глава 10	
Обработка табличной информации	173
10.1. Массивы CLR	173
10.2. Работа с компонентом DataGridView	175
Глава 11	
Добавление компонентов на форму в процессе выполнения приложения	183
11.1. Добавление компонентов и установка их свойств в режиме кода	183
11.2. Обработка событий компонентов, добавленных в процессе выполнения программы.....	185
Глава 12	
Приложения для работы с графикой	189
12.1. Построение графиков.....	189
12.2. Программирование простейшего графического редактора.....	194
12.3. Программирование «движения» графического объекта	200
12.4. Особенности работы с компонентом «коллекция рисунков» – ImageList.....	203

Введение

В данной книге рассматривается использование языка C++/CLI для разработки приложений Windows Forms в среде программирования Microsoft Visual Studio 2008.

Язык C++/CLI является адаптацией C++ для .NET Framework и позволяет довольно быстро и просто разрабатывать приложения с графическим интерфейсом, так как значительная часть кода создается автоматически.

Книга может быть полезна тем, кто обладает некоторыми знаниями языка C++ и желает самостоятельно освоить разработку приложений Windows Forms в среде программирования Microsoft Visual Studio 2008.

В каждой главе присутствует минимальная часть теоретической информации и большое количество заданий для выполнения с подробным пошаговым описанием всех тонкостей разработки интерфейса и написания программного кода.

Рассматриваются особенности объектно-ориентированного программирования на языке C++/CLI: использование ссылочных классов, наследования, перегрузок функций и операций, виртуальных функций и шаблонов. При этом все примеры рассматриваются с использованием визуального программирования.

Большое внимание уделяется технологии разработки приложений Windows Forms в среде программирования Microsoft Visual Studio 2008. Возможности всех компонентов рассматриваются применительно к их использованию для решения различных практических задач.

Выполняя задания, руководствуясь подробным описанием, размещенным в данной книге, вы сможете самостоятельно разработать приложения для работы с текстовой, графической и табличной ин-

формацией, создать свой собственный браузер для просмотра Web-страниц, многостраничное приложение для тестирования знаний и многие другие.

Вы научитесь добавлять компоненты и устанавливать их свойства не только в режиме Конструктора, но и программным образом, сохранять и считывать информацию из файлов, обрабатывать ее, используя дружественный, вами созданный интерфейс.

Глава 1.

Основные понятия объектно-ориентированного программирования

Сегодня нельзя представить себе создание сложных программных продуктов без использования принципов объектно-ориентированного программирования (ООП). Благодаря ООП можно определять и использовать собственные типы данных также как и встроенные. Это позволяет упрощать процесс программирования для решения конкретных прикладных задач.

При этом необходимо абстрагироваться, то есть выделить из множества свойств, характеризующих объект, лишь те, которые важны для данного случая. Например, если речь идет об оценке практичности транспортного средства, то имеют значения его скорость, тип двигателя, пассажировместимость и т.д.

Объектом с точки зрения ООП является реальный конкретный элемент, у которого имеется набор свойств, описывающих его состояние, и набор функций, определяющих его поведение. К примеру, мой автомобиль имеет конкретную марку, номер, цвет, максимальную скорость и многие другие характеристики. Его функции – это способность двигаться с определенной скоростью, разгоняться, тормозить, перемещаться по дорогам определенного типа и т.д.

В то же время каждый автомобиль имеет подобные свойства и функции, характеризующие его как объект класса «автомобиль». Определяя какой-либо класс, мы выделяем и описываем состояние и поведение, характерные для всех объектов, ему принадлежащих. Рассматривая объект класса, мы описываем свойства конкретного, реального предмета: реального автомобиля, книги, стола.

Для защиты от внешних воздействий и повреждений данные и обрабатывающие их функции объединены и скрыты от пользователей. Данный механизм называется инкапсуляцией и позволяет повысить надежность и простоту использования объектов класса. Для доступа к изменению свойств объекта используется интерфейс.

На базе уже описанного класса можно создать новые классы, чтобы использовать уже определенные данные и функции, то есть наследовать их. Как и в живой природе, при этом могут добавлять-

ся новые функции и свойства, а также переопределяться старые.

Также для ООП характерно такое свойство как полиморфизм, когда однотипные, возможно имеющие одинаковое название действия для родственных классов выполняются по-разному. Это достигается за счет описания другой реализации одноименного действия в производном классе.

1.1. Описание класса

Класс представляет собой описание данных, необходимых для представления объекта, и определение функций, выполняющих их обработку. Зачастую класс сравнивают с типом данных, который разрабатывается программистом самостоятельно.

Объявление класса начинается с ключевого слова **class** и выглядит следующим образом [4]:

```
class имя_класса
{закрытые функции и переменные класса
public:
открытые функции и переменные класса
} список_объектов;
```

имя_класса является именем, которое используется при создании объектов класса. *список_объектов* в данном случае может отсутствовать, но знак «;» обязателен.

По умолчанию функции и переменные класса(элементы класса) доступны для использования только внутри класса, то есть имеют статус доступа **private**. Если планируется использовать их вне класса, то необходимо использовать ключевое слово **public**, за которым пишут двоеточие. Ключевое слово **protected** необходимо использовать, если данные и функции класса должны быть доступны только для класса потомка при наследовании.

Определение статуса доступа для элементов [4]:

```
class MC
{ public: // доступно всем
```

```

        // данные, методы, свойства, события
        //-----
protected: // защищенный - доступно только в классе и
        //потомкам при наследовании классов
        // данные, методы, свойства, события
        //-----
private: //собственный - доступно только в классе
        //наиболее ограниченный доступ
        // данные, методы, свойства, события}
< Список переменных>;//не обязателен в объявлении
        //можно объявить объекты позже

```

В объявлении класса ключевые слова **private**, **protected** и **public** могут использоваться в любом порядке и любое число раз.

В качестве примера объявим класс с именем **sss**, в котором используем две закрытых переменных целого типа и четыре функции, две из которых используются для инициализации переменных класса, одна – для расчета их суммы, а другая – для определения разности.

```

ref class sss
{
int x,y;    // по умолчанию private
public:
void getx(int x1)  { x=x1;}    //определение функции
void gety(int y1)  { y=y1;}    //определение функции
int  summa();      // прототип функции
int  razn();
};

```

Так как мы будем использовать данный класс при разработке приложения типа Windows Form в среде программирования Visual Studio 2008, то перед ключевым словом **class** автоматически прописывается **ref**. Это значит, что класс ссылочного типа, и его объекты будут размещаться в выделенной средой CLR (Common Language

Runtime – общезыковая среда исполнения) динамической памяти. В данном случае нет необходимости в удалении ненужного объекта, так как память автоматически от него очищается. При этом адреса других объектов изменяются, и в качестве указателя используются отслеживаемые дескрипторы, которые обозначаются символом ^, а для выделения памяти применяется утилита *gcnew* [1].

Объявленный выше класс имеет две закрытые переменные, которые доступны для использования только внутри класса *sss*. Для изменения их значений и использования в расчетах вне класса необходимо использовать открытые функции *getx()*, *gety()*, *summa()*, *razn()*, которые могут вызываться из любой части программы, использующей класс *sss*.

Помещение данных класса в область *private*, а функций – в *public* позволяет реализовать принцип инкапсуляции и упростить процесс программирования, так как при изменении действий, выполняемых функциями, необходимо внести корректировку только в тело функции, а не в программу пользователя.

Функции класса *getx()*, *gety()* определены при объявлении класса и по умолчанию считаются встраиваемыми. Рациональное использование таких функций способствует увеличению скорости выполнения программы и уменьшению объектного кода.

Функции *summa()*, *razn()* объявлены в *sss*, а определены в отдельном файле *sss.cpp*, который создается средой автоматически, имеет по умолчанию такое же имя как и класс, но с расширением *.cpp*.

При определении функции вне класса следует использовать следующий формат [5]:

```
Тип_возвр_значения имя_класса: :имя_функции  
(список_параметров)  
{  
// тело функции  
}
```

Здесь *имя_класса* – это имя класса, которому принадлежит определяемая функция. Для обозначения принадлежности

определяемой функции классу используется оператор два двоеточия (::), который пишется после имени класса перед именем функции.

имя_класса: :имя_функции

В рассматриваемом случае при определении функций *summa()* и *razn()* класса *sss* это выглядит следующим образом:

```
#include "sss.h" //подключение заголовочного
                  //файла с объявлением класса
int sss :: summa()
{ return x+y; }

int sss :: razn()
{ return x-y; }
```

Если есть необходимость использования функции, определенной вне класса, как встраиваемой, то нужно использовать ключевое слово **inline**.

```
inline int sss :: summa()
{ return x+y; }
```

1.2. Создание и использование объектов

Для создания объекта класса можно использовать запись:

имя_класса имя_объекта;

Например,

```
sss ob1, ob2; //объекты класса sss
sss x[80];    // массив из 80 объектов класса sss
```

При объявлении создается собственно объект и выделяется память. Этого не происходит при объявлении класса.

После создания объекта можно использовать открытые функции класса с помощью оператора точка(.):

имя_объекта.имя_функции (список_параметров);

Например, применительно к рассмотренному выше классу это выглядит так:

```
ob1.getx(5); //вызов функции getx(5) для объекта ob1
```

Если в разных базовых классах имеются функции с одинаковыми именами, то целесообразно использовать следующую запись с непосредственным указанием имени класса перед именем объекта [4]:

имя_объекта.имя_класса::имя_функции(список_параметров);

то есть вышеописанный вызов функции `getx(5)` для объекта `ob1` будет выглядеть следующим образом:

```
ob1.sss::getx(5);
```

Для каждого из созданных объектов класса переменные класса имеют собственное значение, то есть в результате выполнения следующих действий переменная *x* объекта ***ob1*** принимает значение 5, а переменная *x* объекта ***ob2*** принимает значение 7.

```
ob1.getx(5);
```

```
ob2.getx(7);
```

Если мы попытаемся обратиться к закрытой переменной или функции вне класса, то возникает ошибка.

```
ob1.x=10;    ob2.x=7; //ОШИБКА!
```

Устранить проблему поможет открытие переменной или функции:

```
ref class sss
{
public:
    int x,y; // теперь открыты!
    void getx(int x1) { x=x1; } //определение функции
    void gety(int y1) { y=y1; } //определение функции
    int summa(); // прототип функции
    int razn();
};
```

В данном случае мы можем использовать запись:

```
ob1.x=10;    ob2.x=7; // Ошибки нет!
```

Однако рекомендуется открывать функции, а не данные.

1.3. Конструкторы и деструкторы

В рассмотренном выше примере для инициализации данных класса использовались функции *getx()*, *gety()*. Однако, как правило, для этих целей используется специальная функция класса, называемая конструктором. Данная функция имеет имя, совпадающее с именем класса, и вызывается каждый раз при объявлении объекта класса или создании объекта с помощью *gcnew*. Отличительной чертой данной функции является то, что она не имеет возвращаемого значения.

Определим знакомый уже класс *sss* с использованием конструктора для инициализации переменных класса *x, y*.

```
ref class sss
{
int x,y;
// по умолчанию private
public:
sss(); //конструктор по умолчанию
int summa();
// прототип метода:
int razn();
};
```

Определение функций размещено в отдельном файле:

```
#include "sss.h" //подключение заголовочного
                //файла с объявлением класса

sss::sss()
{x=4;
 y=7;
}
```

```
int sss :: summa()
{ return x+y; }
```

```
int sss :: razn()
{ return x-y; }
```

Используем конструктор и функции класса в обработке события «Нажатие на кнопку»:

```
private: System::Void button2_Click(System::
Object^ sender, System::EventArgs^ e)
{
    sss ob1;
    int n,m;
    n=ob1.summa(); \\ равно 11
    m=ob1.razn(); \\ равно -3
    . . .
}
```

В соответствии с вышесказанным при объявлении объекта класса **sss** **ob1** вызывается конструктор данного класса. В результате создается объект **ob1** класса **sss**, переменная **x** получает значение **4**, а переменная **y** значение **7**.

Рассмотренный выше конструктор называется конструктором по умолчанию. При создании любого объекта класса данным класса присвоятся значения, указанные в теле конструктора.

Конструктору данного вида нельзя передавать аргументы. Решить указанную проблему можно с помощью конструктора с параметрами. При этом параметры задаются в качестве аргументов в объявлении и определении конструктора. Рассмотрим на том же примере, заменив конструктор по умолчанию на конструктор с параметрами.

```
ref class sss
{ // по умолчанию private
    int x,y;
public:
```

```

sss(int a, int b); //конструктор с параметрами
int summa();
// прототип метода:
int razn();
};

```

Определение функций в отдельном файле:

```

#include "sss.h"
sss::sss(int a, int b)
{
    x=a;
    y=b;
}

int sss :: summa()
{
    return x+y;
}

int sss :: razn()
{
    return x-y;
}

```

Теперь в обработчике события события «Нажатие на кнопку» используем следующую запись:

```

private: System::Void button2_Click(System::
Object^ sender, System::EventArgs^ e)
{
    sss ob1(4,5);
    int n,m;

    n=ob1.summa(); \\ равно 9
    m=ob1.razn(); \\ равно -1
    . . .
}

```

В обработчике события объявляется объект класса *sss ob1*. Здесь конструктор класса *sss* имеет два параметра. Значения, записанные в круглых скобках, используются для инициализации переменных *x* и *y*. Они передаются параметрам *a* и *b* конструктора и затем присваиваются переменным *x* и *y* соответственно как при обычном вызове функции. В результате 4 передается в *x*, а 5 в *y*. Число аргументов

может быть любым и зависит от возможностей компилятора.

В качестве параметров могут быть использованы не только константы, но и выражения с переменными. Забегая вперед, используем в обработчике события ввод исходных данных в текстовые поля:

```
private: System::Void button2_Click(System::
Object^ sender, System::EventArgs^ e)
{
    int nc,mc,ac,bc;
    ac=Convert::ToInt16(textBox1->Text);
    bc=Convert::ToInt16(textBox2->Text);
    sss obl(ac,bc);
    nc=obl.summa(); \\ равно 9
    mc=obl.razn(); \\ равно -1
    . . .
}
```

Еще одним видом конструктора является конструктор со списком инициализации. Рассмотрим его использование на примере конструктора класса *bs*, имеющего в качестве данных одну переменную *x* целого типа.

В случае применения конструктора с параметрами запись выглядела бы так:

```
bs::bs (int n) {x=n;}
```

Конструктор со списком инициализации в данном случае имеет следующий вид:

```
bs(int n):x(n) {}
```

Важным замечанием является то, что определение данного конструктора должно находиться в теле определения класса, то есть в файле **bs.h**. Значения данных класса указываются в списке инициализации как часть заголовка функции. В приведенном примере переменная *x* инициализируется значением *n*. Операцию присваивания здесь не используют.

В следующем примере используется конструктор для инициализации трех переменных класса. Важно: список инициализации отделен запятой от списка параметров, инициализаторы перечисляются через запятую, тело функции пустое.

```
ref class pass
{int x;int z; double m;
public:
    pass(int a,int b,double c):x(a),z(b),m(c) {}
};
```

В отличие от конструктора, вызываемого при создании объекта, при удалении объекта вызывается деструктор. При использовании классов ссылочного типа применение деструктора неактуально из-за автоматической очистки памяти от ненужных объектов. Однако его можно применить для принудительного удаления объекта, вывода сообщения, закрытия файла и т.д.

Имя деструктора, как и конструктора, совпадает с именем класса, но перед именем записывается символ ~ (тильда). Пример: ~sss(); В отличие от конструктора деструктор не может иметь параметры.

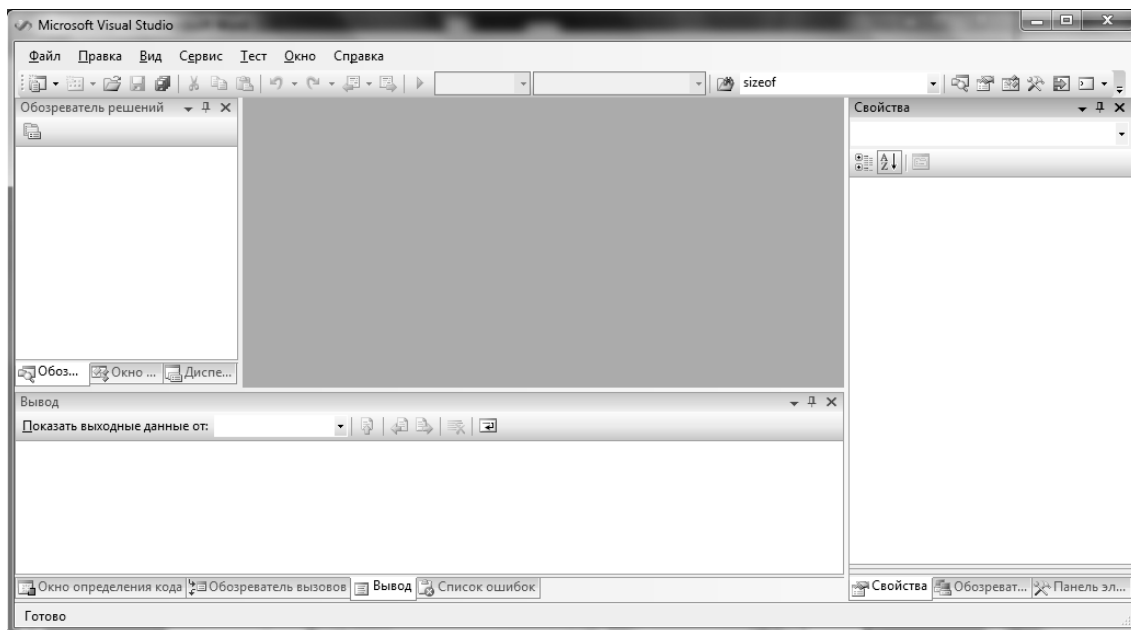
В теле конструктора и деструктора могут выполняться любые операции. Однако принято использовать данные функции по назначению.

Глава 2.

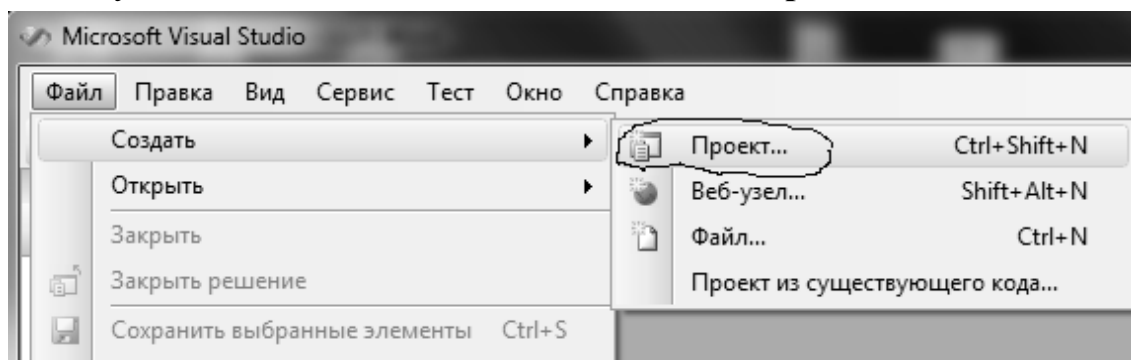
Технология разработки приложения Windows в среде Visual Studio 2008 на языке C++/CLI

2.1. Создание проекта

После загрузки Visual Studio 2008 перед вами будет следующее окно:

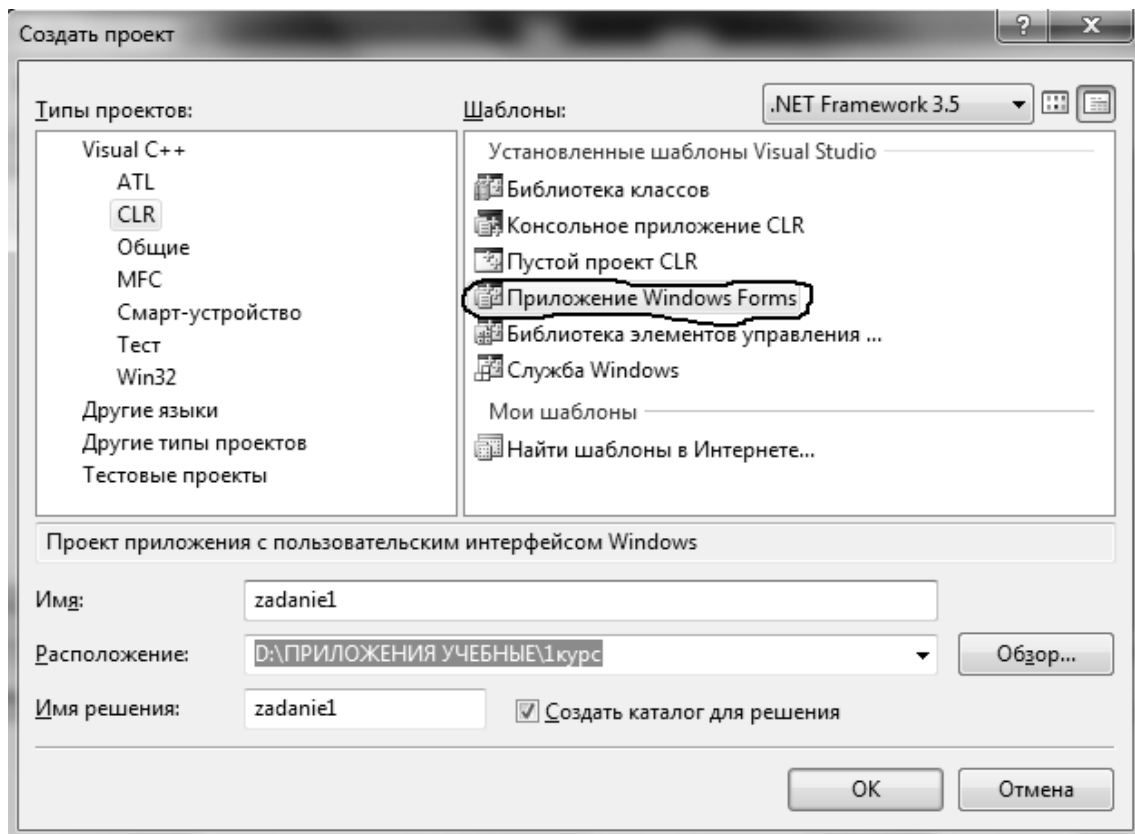


Чтобы начать разработку приложения, необходимо прежде всего создать проект. Для этого выполните ряд действий с помощью пунктов главного меню: **Файл/Создать/Проект...**

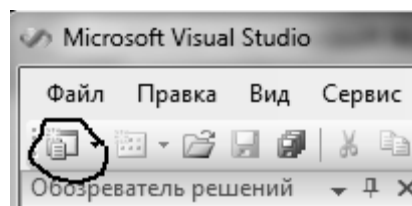


Откроется диалоговое окно, где нужно выбрать тип проекта: **Visual C++/CLR/Приложение Windows Forms**, папку, где будет располагаться приложение (кнопка **Обзор...**), ввести имя и нажать кнопку **ОК**.

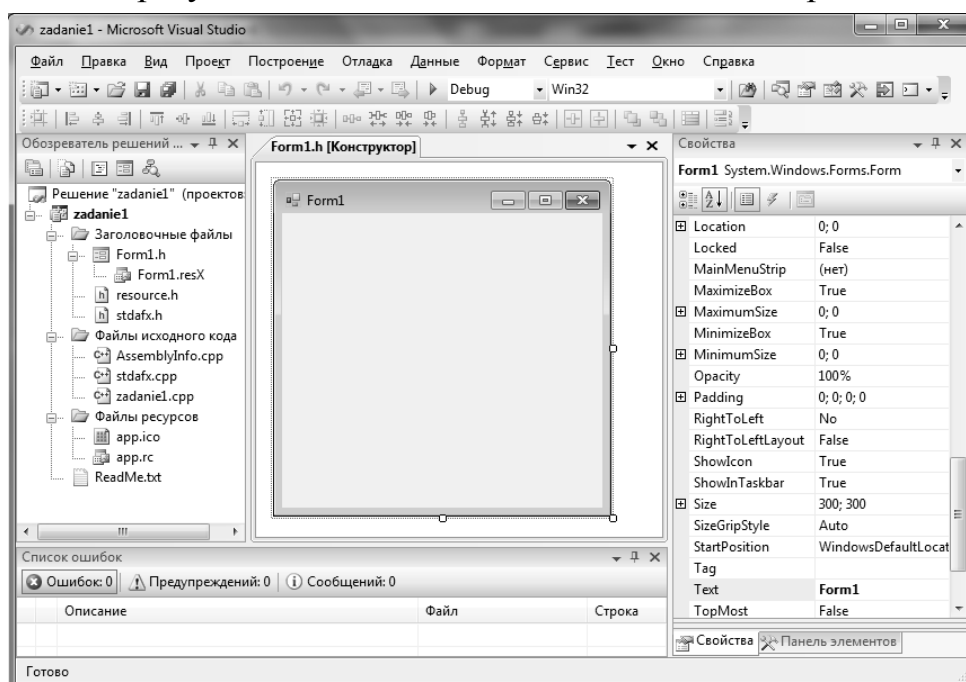
Если по умолчанию в типах проектов установлен Visual C# или Visual Basic, то выбрать Visual C++ можно в группе **Другие языки**.



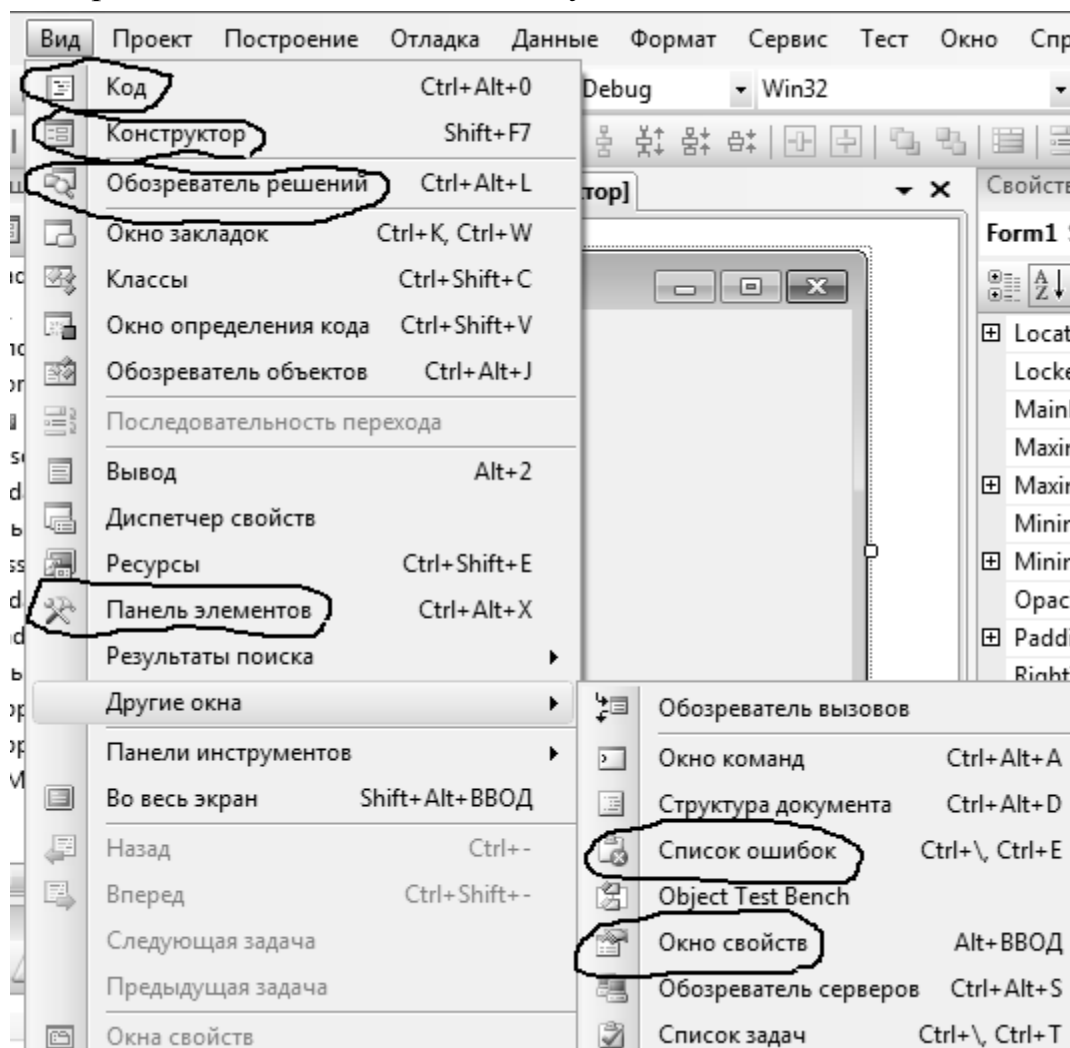
Для этой же цели можно применить кнопку быстрого вызова **Создать проект**.



В результате выполненных действий окно примет вид:



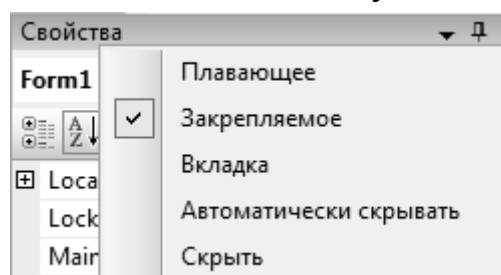
Рассмотрим только основные окна, отображенные на экране: **Свойства**, **Обозреватель решений**, **Панель элементов**, **Конструктор**, **Код**, **Список ошибок**. В случае, если вы закрыли какое-то из них, для открытия можно воспользоваться пунктом главного меню **Вид**.



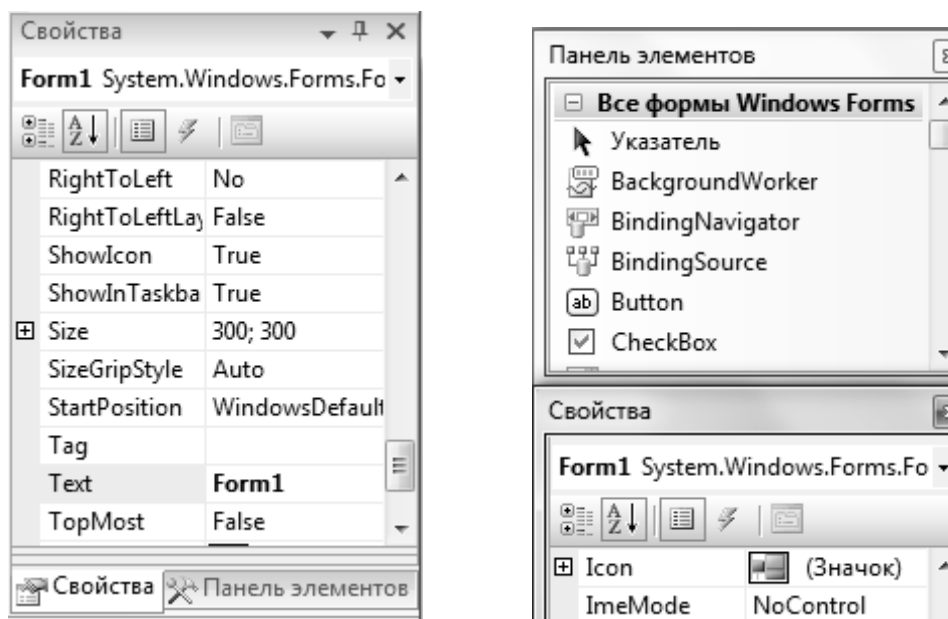
Можно воспользоваться и панелью быстрых кнопок.



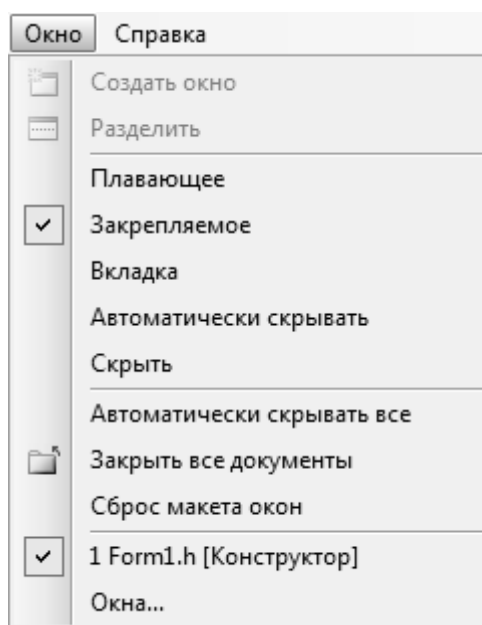
У всех окон имеется контекстное меню, которое появляется при щелчке правой кнопкой мыши по заголовку окна.



В случае выбора из контекстного меню варианта **Закрепляемое** можно перетащить с помощью левой кнопки мыши заголовки одного окна на другой и расположить окна в удобном для вас виде. Например, так:



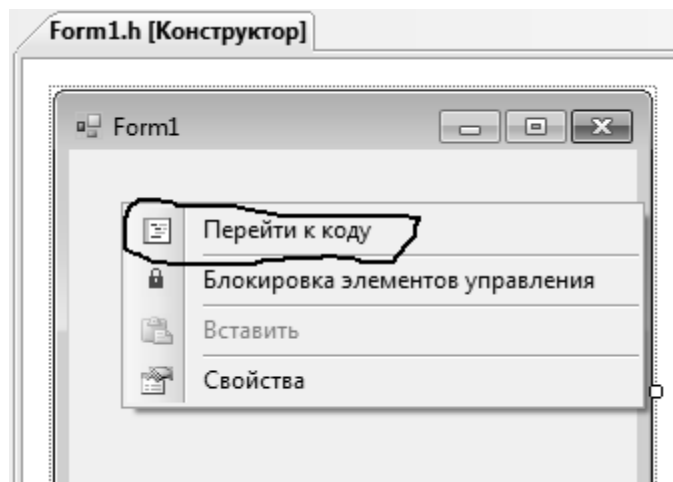
Для работы с окнами удобно пользоваться также пунктом главного меню **Окно**.



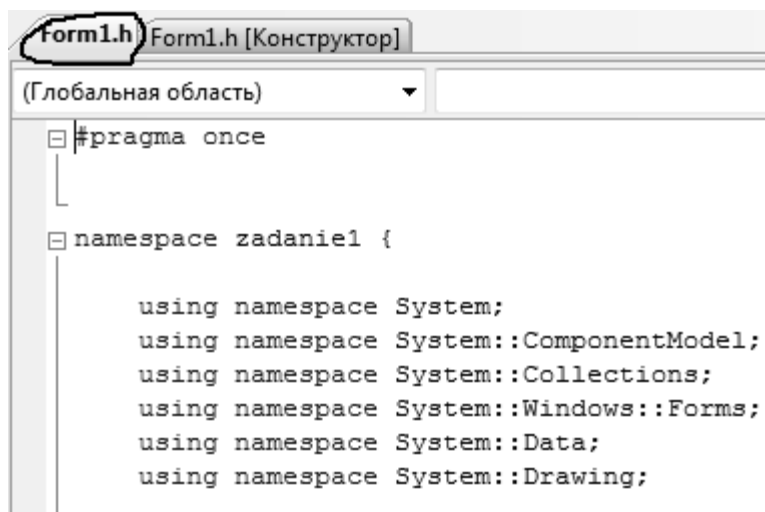
Одно и то же действие в среде программирования можно выполнить несколькими способами. Здесь рассматриваются только некоторые из них.

Так перейти к программному модулю формы (Form1.h по умолчанию) можно и с помощью главного меню (**Вид/Код**), и с помощью

контекстного меню формы.



В результате рядом с вкладкой **Form1.h (Конструктор)** откроется вкладка **Form1.h**, содержащая программный код, описывающий окно приложения Windows Forms.



Рассмотрим код для пустой формы.

#pragma once — препроцессорная директива, контролирующая однократность подключения данного исходного файла при компиляции.

Класс **Form1** определен в собственном пространстве имен:

namespace zadanie1 — собственное пространство имен, в котором определен класс **Form1**. **zadanie1** — имя пространства имен, совпадающее с именем проекта, которое мы указали при создании приложения. В случае если в различных сборках имеются типы с одинаковыми именами, то они уточняются именно с помощью пространств имен, к которым принадлежат.

Чтобы обратиться к любому имени вне пространства имен, в ко-

тором они объявлены, необходимо перед таким именем использовать префикс **имяПроекта:: (std::cin)** или перед его использованием записать **using namespace имя_пространства_имен**; указав к какому пространству относится имя. Пространство имен сравнивают с контейнером для переменных, функций, классов и т.д.

По умолчанию в коде формы используются пространства имен библиотеки .NET, содержащие классы, выполняющие следующие действия:

using namespace System – определяют типы данных для приложений CLR, описывают события и обработчики событий, исключения, поддерживают функции общего применения.

using namespace System::ComponentModel – обеспечивают возможность использования компонентов графического интерфейса.

using namespace System::Collections – организуют данные (определяют списки, очереди, словари, стеки).

using namespace System::Windows::Forms – обеспечивают применение в приложениях средств Windows Forms.

using namespace System::Data – реализуют доступ к источникам данных с помощью набора компонентов ADO.NET.

using namespace System::Drawing – поддерживают графические операции на форме или компоненте.

После указания необходимых для работы приложения пространств имен идет определение класса Form1.

```
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:
    Form1(void)
    {
        InitializeComponent();
        //
        //TODO: добавьте код конструктора
        //
    }

protected:
```

Данный класс имеет квалификатор **ref**, то есть при работе приложения будет использована автоматическая сборка мусора. Кроме того класс **Form1** является производным от класса **Form**, определен-

ном в пространстве имен **System::Windows::Forms**, и наследует все свойства и события класса **Form**. По умолчанию программный модуль первой формы приложения получает имя **Form1.h**, второй – **Form2.h** и т.д.

Далее следует раздел кода, который автоматически изменяется при изменении вида формы в режиме Конструктора. Как только на форму добавляется компонент, он сразу прописывается. Его не рекомендуется корректировать самостоятельно в окне редактора кода. Только опытные пользователи могут вносить туда поправки.

Для определения вида окна приложения и компонентов, находящихся на форме конструктор вызывает функцию **InitializeComponent()**. Как только на форму добавляется компонент, он сразу прописывается в теле данной функции. Изменение его свойств также отражается там. При удалении компонента с формы его код удаляется также автоматически. Форма является контейнером для хранения всех элементов, расположенных на ней.

Код функции **InitializeComponent()** при пустой форме имеет вид:

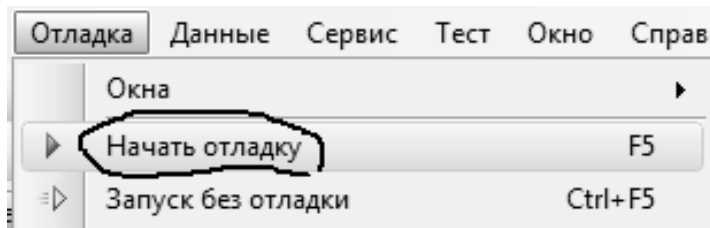
```
void InitializeComponent(void)
{
    this->SuspendLayout();
    this->AutoScaleDimensions = System::Drawing::
    SizeF(6,13);
    this->AutoScaleMode = System::Windows::Forms::
    AutoScaleMode::Font;
    this->ClientSize = System::Drawing::Size(284,262);
    this->Name = L"Form1";
    this->Text = L"Form1";
    this->ResumeLayout(false);
}
```

Так как компонентов на форме нет, то описываются только свойства формы. Назначение каждого из свойств легко узнать из всплывающей подсказки, подведя курсор к интересующему свойству.

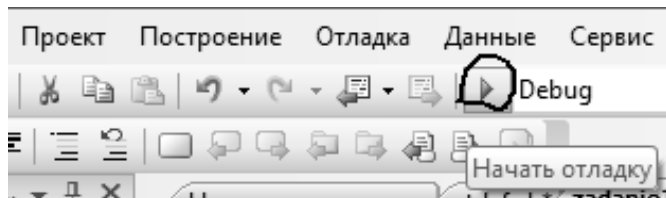
```
this->Name = L"Form1";
this->property System::String ^System::Windows::Forms::Control::Name
this->Возвращает или задает имя элемента управления.
```

Запустить приложение на выполнение одновременно с отладкой

и построением можно с помощью команды главного меню,



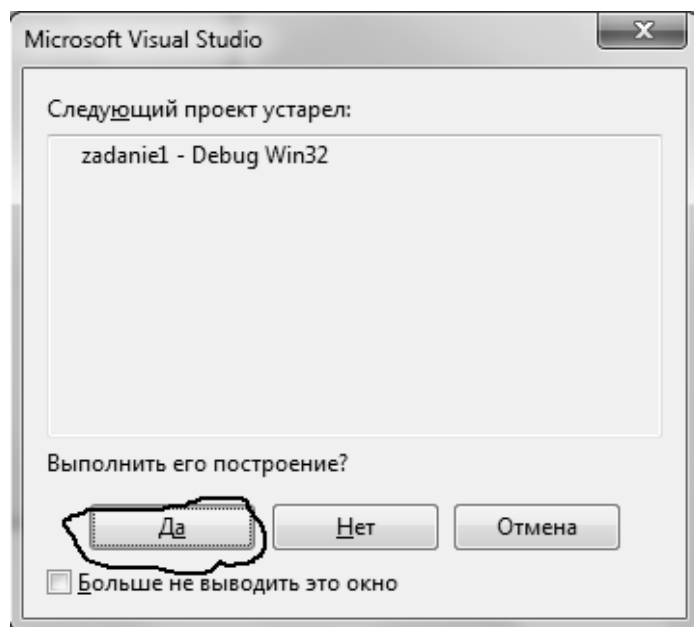
быстрой кнопки **Начать отладку** на панели инструментов,



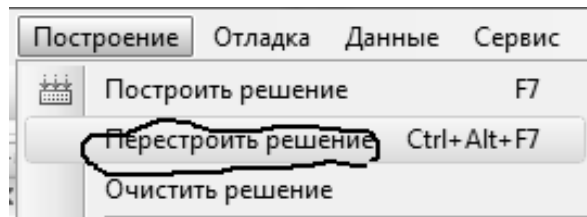
клавиши на клавиатуре **F5**.

Если выбрать конфигурацию решения **Release** вместо **Debug**, то создается исполняемый файл приложения, который можно запускать на компьютере, где нет установленной **Microsoft Visual Studio**. Взять данный файл можно в папке **Release** созданного приложения.

При выполнении любого из вариантов запуска появится сообщение. В ответ на предложение построения рекомендуется согласиться.



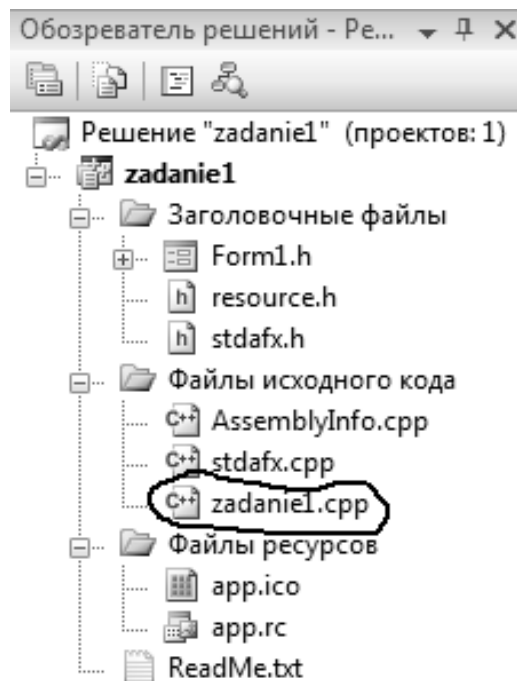
Если такое сообщение не выводится, лучше перестроить проект перед запуском, так как иногда не отражаются внесенные в него изменения, возникают ошибки. Сделать это проще с помощью команды главного меню.



В рассматриваемом случае в результате мы увидим пустую форму, закрыть которую можно с помощью кнопки закрытия в правом верхнем углу.



Запуск приложения начинается с выполнения функции **main()**, которая находится в главном файле проекта, имеющем имя, совпадающее с именем проекта и расширение **.cpp**. В данном случае это файл **zadanie1.cpp**. Открыть его можно двойным щелчком по имени файла в окне **Обозревателя решений**.



В результате появится вкладка **zadanie1.cpp** рядом с кодом формы **Form1.h**, содержащая код:

```

#include "stdafx.h"
#include "Form1.h"
using namespace zadanie1;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
    // Включение визуальных эффектов Windows XP до
    создания каких-либо элементов управления
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault
    (false);

    // Создание главного окна и его запуск
    Application::Run(gcnew Form1());
    return 0;
}

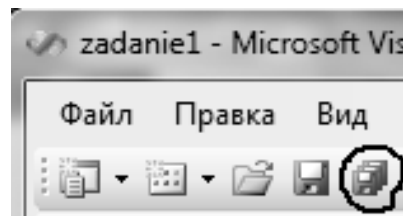
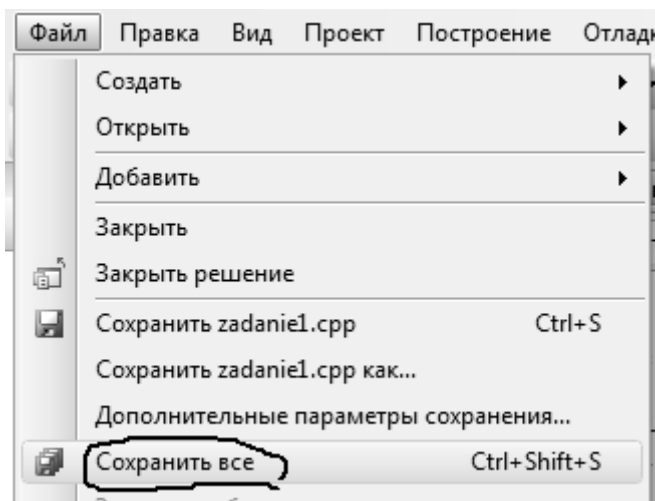
```

С помощью директивы препроцессора **#include "stdafx.h"** подключается файл для стандартных системных включаемых файлов или файлов, включаемых для конкретного проекта, часто используемых, но редко изменяемых. Открыв файл **stdafx.h** из окна **Обозревателя решений**, можно добавить туда различные директивы препроцессора, которые будут доступны во всех файлах проекта. К примеру, туда можно перенести следующую строчку файла **zadanie1.cpp** – директиву препроцессора, подключающую файл формы **Form1.h** – **#include "Form1.h"** или добавить другие необходимые вам директивы.

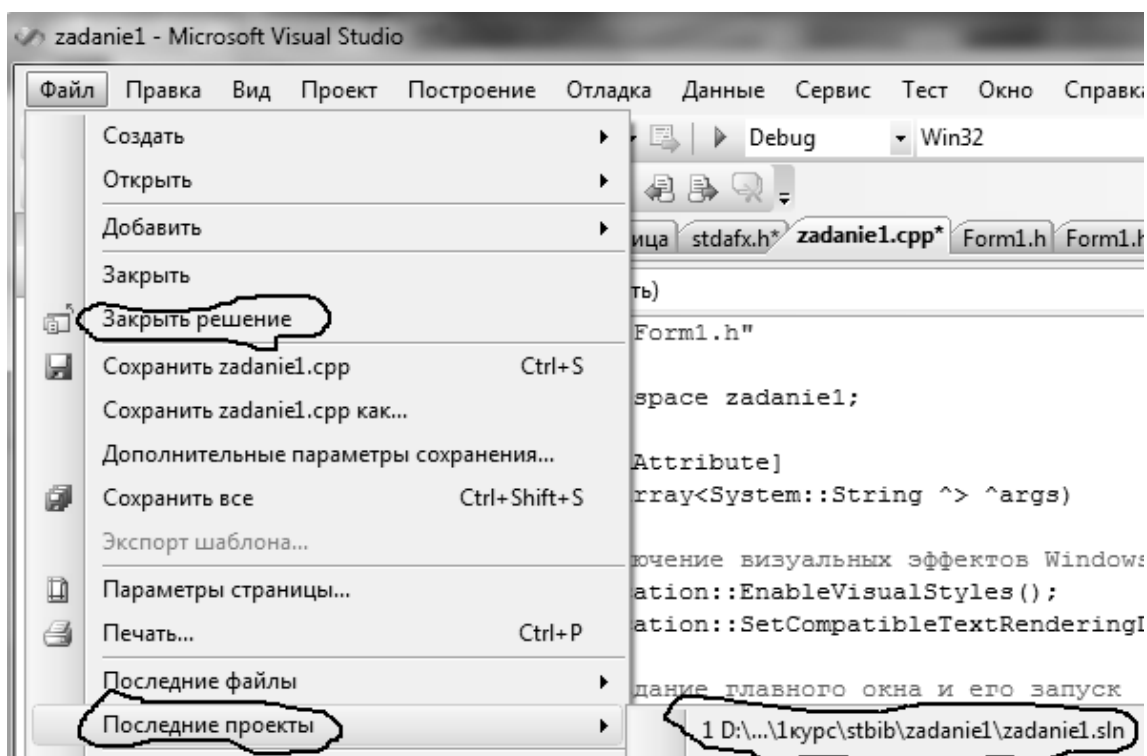
Также в данном файле используется запись **using namespace zadanie1;**, поскольку функция **main()** обязательно должна быть в пространстве имен, чтобы компилятор мог распознать ее.

Назначение функций, используемых в функции **main()** легко узнать по всплывающим подсказкам. Все они определены в классе **Application** пространства имен **System: :Windows: :Forms** и являются основой приложений **Windows Forms**. Форма, указанная в качестве аргумента функции **Run()** всегда запускается первой.

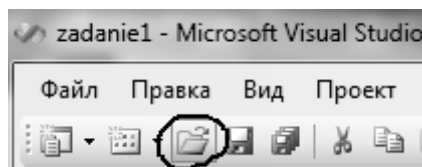
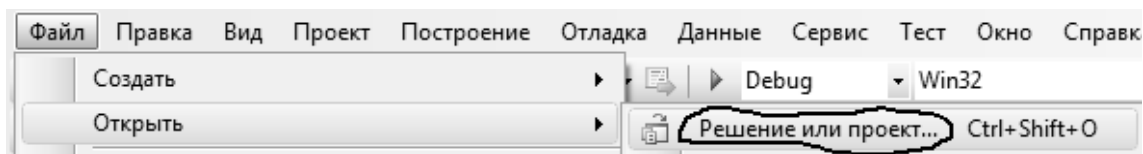
Сохранить созданный проект можно с помощью команды главного меню или быстрой кнопки на панели инструментов.



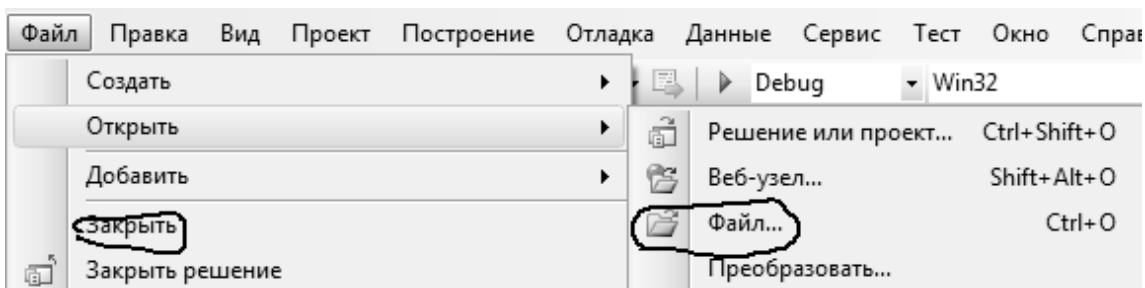
Для закрытия всех проектов можно выполнить команду главного меню **Файл/Заккрыть решение**, для открытия проекта с которым работали недавно – использовать команду **Файл/Последние проекты**, где выбрать нужный проект из списка.



Если нужного проекта нет в списке последних, то опять же можно использовать команду главного меню или быструю кнопку. В открывшемся диалоговом окне в папке проекта выбрать файл с расширением **.vcproj**.



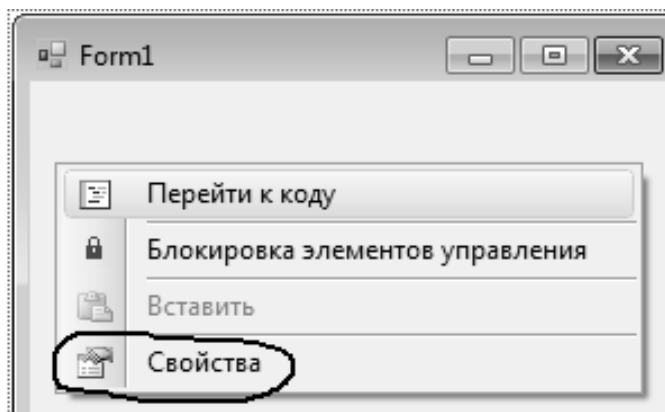
Подобные действия выполняют, когда нужно открыть или закрыть отдельные файлы проекта.



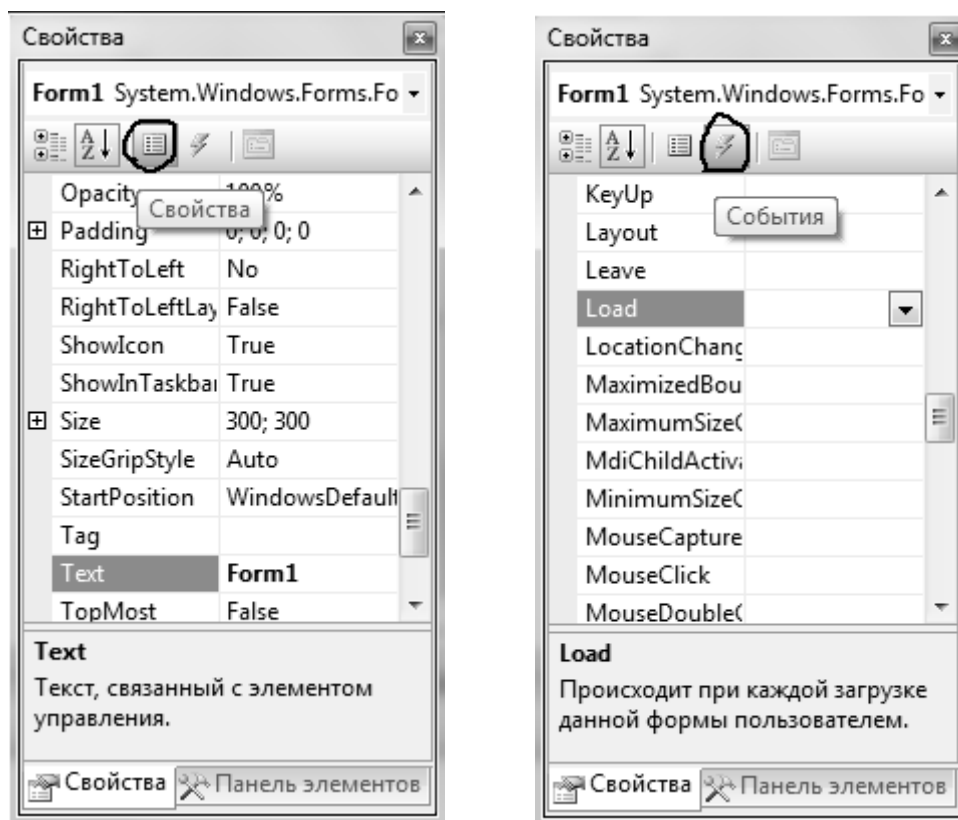
2.2. Окно сведений об объекте

Каждый компонент, включая форму, является объектом соответствующего класса, который характеризуется набором свойств и методов, выполняемых в ответ на определенное событие.

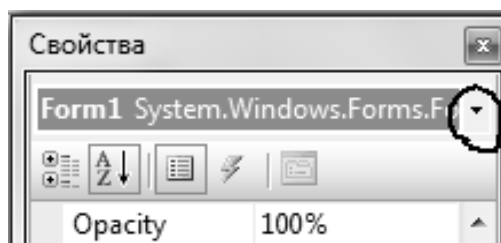
Чтобы получить доступ к свойствам и событиям конкретного компонента, необходимо его выделить в режиме **Конструктора**. Все необходимые сведения отобразятся в окне **Свойства**. Если данное окно скрыто, открыть его можно способом, описанным ранее, или, воспользовавшись контекстным меню данного компонента.



Данное окно имеет две вкладки: **Свойства** и **События**, переключение между которыми осуществляется щелчком левой кнопкой мыши по соответствующей пиктограмме.

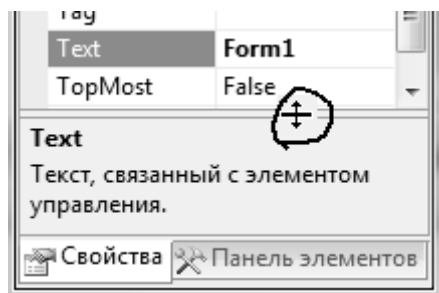


Объект можно выделить не только щелчком по нему, но и с помощью окна **Свойства**, используя выпадающий список вверху окна. Набор свойств и событий будет относиться уже к выбранному объекту. Это особенно полезно, если объект не виден.



Описание каждого свойства можно увидеть в нижней части окна, настроив ширину для вывода сообщения самостоятельно, подведя курсор мыши к разделяющей полосе так, чтобы он приобрел вид двунаправленной стрелки, и перетащить границу, нажав левую кнопку

мышь.



На вкладке **События** находится список событий, возможных для выделенного компонента. При наступлении какого-либо события могут выполняться определенные действия. Эти действия описываются в обработчике события, который представляет собой функцию с автоматически сформированным заголовком и телом, в котором записываются необходимые в каждом конкретном случае команды, описывающие реакцию на событие, произошедшее с компонентом.

Проще всего добавить обработчик выбранного события с помощью двойного щелчка в поле справа от события.



В результате откроется окно кода формы, курсор будет находиться именно в том месте, где нужно писать код обработчика. Кроме того у каждого компонента есть, так называемые, события по умолчанию. Так создать обработчик нажатие на кнопку(**Click**) можно, выполнив двойной щелчок по ней. При двойном щелчке по форме создастся обработчик события, происходящего при загрузке формы (**Load**)

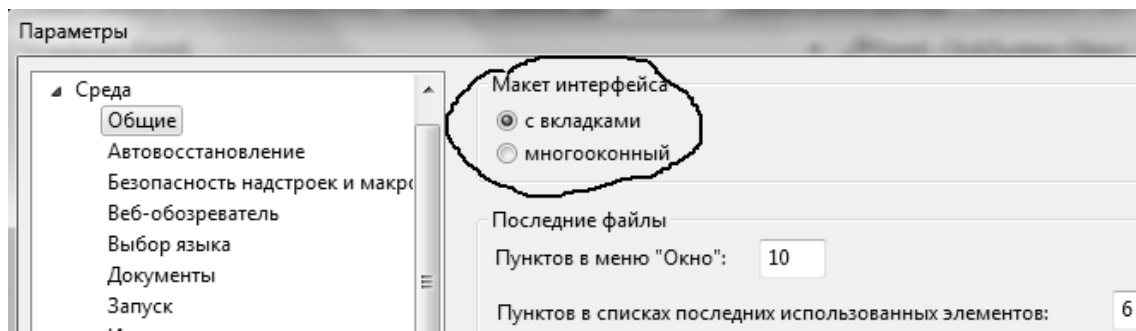
Исследуйте назначение кнопок под заголовком окна **Свойства**. Расположите свойства по алфавиту и категориям, те же действия произведите в окне **События**. Обратите внимание на всплывающие подсказки и характеристику свойств-событий внизу окна.

Закройте окно **Свойства** и откройте его из контекстного меню формы или файла.

2.3. Управление окнами документов

При работе в Microsoft Visual Studio 2008 можно выбрать удобный для себя режим работы с окнами. Для этого необходимо воспользоваться командой главного меню: **Сервис/ Параметры/ Среда/ Общие** и выбрать нужное.

Если удобнее работать с документами, размещенными на вкладках, то выбирайте **С вкладками**. При выборе Многооконного режима для отображения каждого документа будет использовано отдельное окно.



2.4. Редактор кода и режим проектирования

Очень полезной функцией редактора кода является суфлер кода, который значительно упрощает набор кода за счет подсказки, которая появляется после набора стрелки (->). При этом отображаются все элементы класса. Если набрать первые буквы нужного элемента, то список сузится. Теперь достаточно выбрать его щелчком левой кнопки мыши.

Принудительный вызов суфлера кода обеспечивает комбинация клавиш <Ctrl>+<пробел>. Это позволяет выбрать имя объекта (компонента) из возможных. Чтобы увидеть все свойства и методы формы достаточно набрать **this->**.

Редактор кода можно настроить, используя ряд команд главного меню:

1. **Окно/Разделить** – обеспечивает видимость разных частей одного документа одновременно. Чтобы удалить разделение используйте команду **Окно/Снять разделение** или двойной щелчок по полюсе разделения.

2. Для создания копии документа – **Окно/Создать окно**.

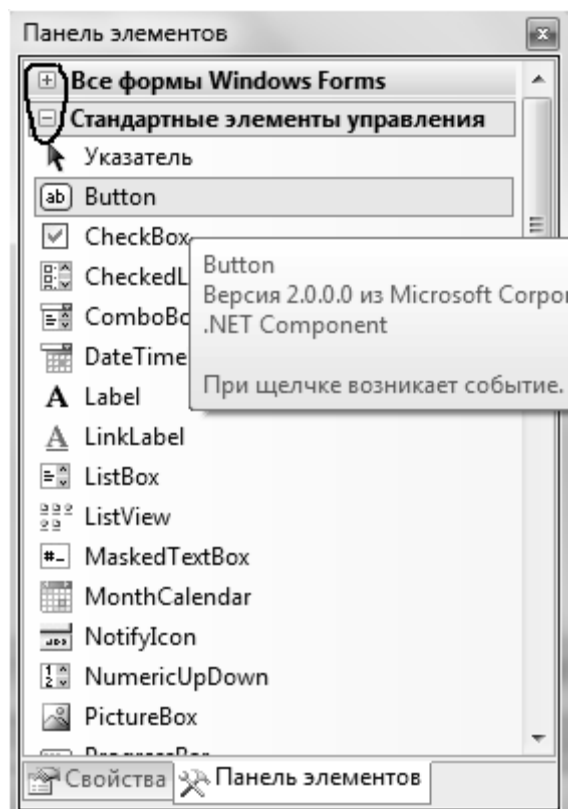
3. Для настройки параметров шрифта – **Сервис/Параметры/Среда/Шрифты и цвета.**

4. Для общих настроек текстового редактора – **Сервис/Параметры/Текстовый редактор/Общие.**

2.5. Конструктор форм

Изменять внешний вид формы, компонентов, их свойства удобно в режиме конструктора, который отображается при открытии **Form1.h[Конструктор]**. Если при открытии приложения данное окно не отображается, можно использовать **Обозреватель решений**, кликнув дважды левой кнопкой мыши по файлу **Form1.h** в списке заголовочных файлов.

Форма является контейнером для компонентов, каждый из которых при помещении на форму по умолчанию наследует многие свойства формы. Компоненты добавляются на форму из **Панели элементов**, разбитой на функциональные группы, содержимое которых можно открывать и сворачивать, нажимая на значок слева от названия группы.



Можно изменить состав Панели элементов, используя команду главного меню: **Сервис/Выбрать элементы панели элементов** и, отметив нужные элементы галочками.

Существует несколько способов добавления компонента на форму:

- щелкнуть по компоненту на палитре компонентов, а затем в том месте формы, где необходимо его расположить;
- перетащить компонент на форму с палитры;
- дважды щелкнуть по компоненту на палитре.

Выделить несколько компонентов можно так:

- удерживая клавишу <Shift>(<Ctrl>), щелкать мышью на требуемых компонентах;
- протянуть указатель мыши, удерживая ее левую кнопку, над выделяемыми элементами.

Снять групповое выделение компонентов можно щелчком мыши вне выделенного пространства.

Размер формы и компонентов можно изменять, используя перетаскивание маркеров по углам и сторонам объектов с помощью левой кнопки мыши. Более точного изменения размеров можно добиться, применив сочетание нажатой клавиши <Shift> и нужных клавиш со стрелками.

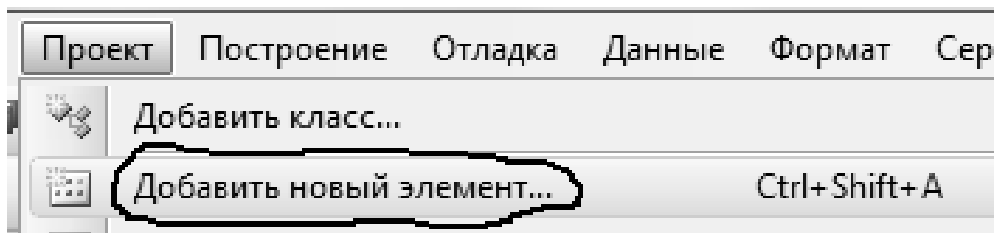
Переместить компонент можно с помощью кнопки мыши или сочетания нажатой клавиши <Ctrl> и клавиш со стрелками.

После размещения компонентов на форме изменяют их свойства, используя окно **Свойства**, и создают обработчики событий.

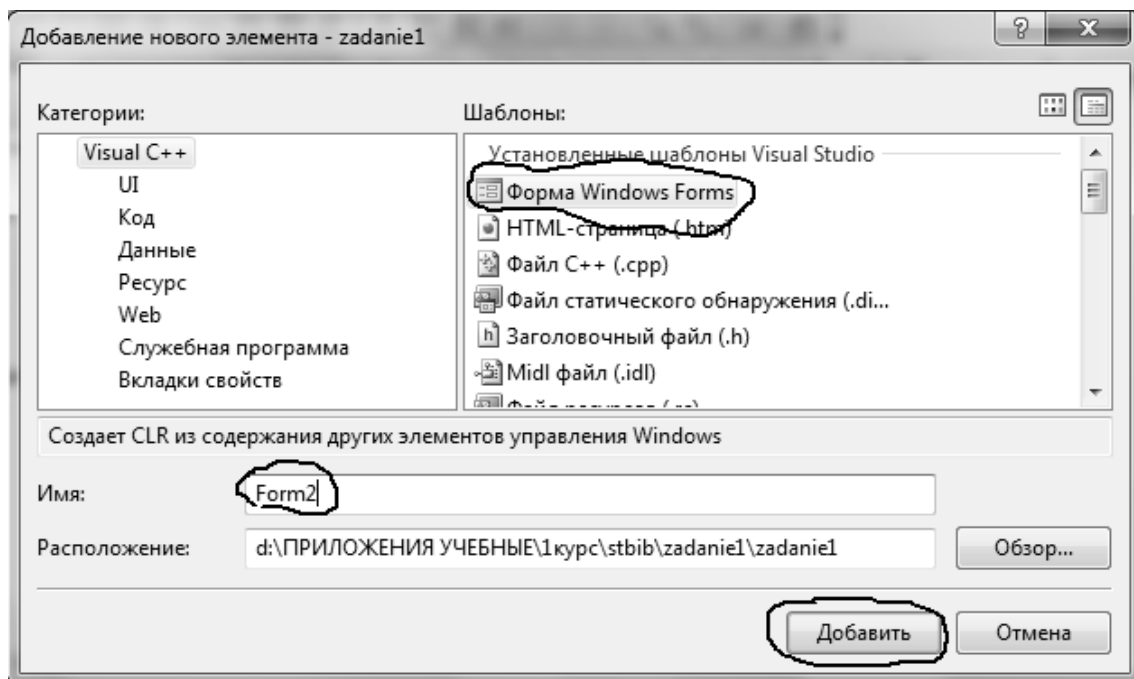
2.6. Использование нескольких форм

Главной формой проекта является та, которая добавляется в проект при его создании. Имя этой же формы прописывается в функции **main()**. Однако можно использовать множество форм, организовав вызов на исполнение любой из них в обработчиках событий.

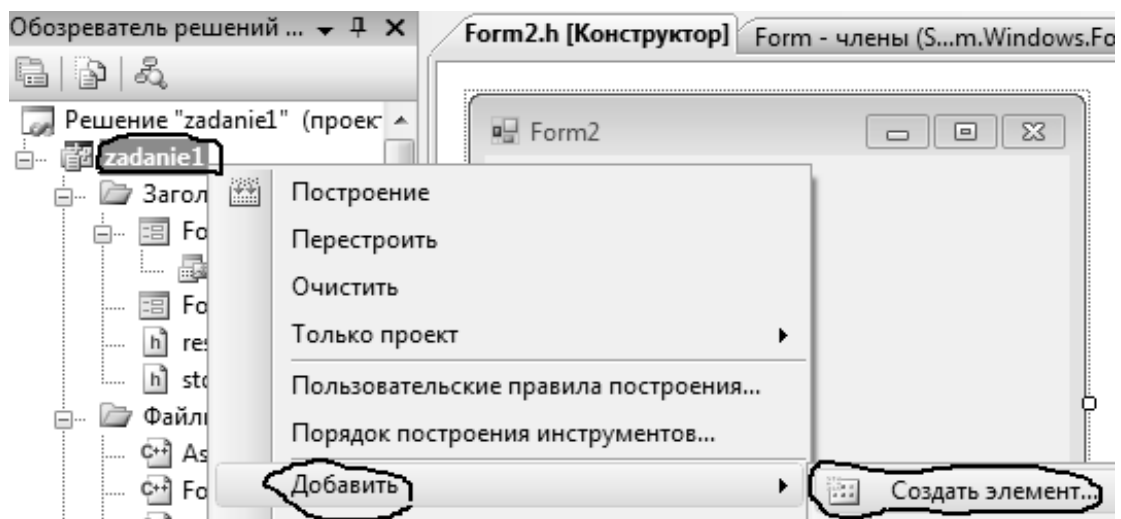
Чтобы добавить в проект новую форму нужно воспользоваться пунктом меню **Проект/Добавить новый элемент**:



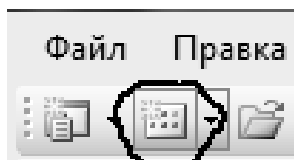
В появившемся диалоговом окне нужно выбрать категорию **Visual C++** и шаблон **Форма Windows Forms**, а также задать расположение имя новой формы **Form2**. Расположение формы предлагается автоматически в той же папке, где находится проект.



Можно также использовать контекстное меню окна **Обозреватель решений** и выбрать в нем опцию **Добавить/Создать элемент...**



Также можно использовать кнопку на панели инструментов.



Отобразите код новой формы и изучите команды из контекстного меню. Обратите внимание на изменение в окне **Обозреватель решений**.

Для выполнения поставленной задачи поместите на каждую из созданных вами форм по две кнопки **Button** любым способом. Разместите их так, как показано на рисунке и измените свойство **Text**, предварительно выделив их и воспользовавшись окном **Свойства**.



Обратите внимание, как изменился при этом код форм.

Чтобы обеспечить открытие второй, созданной вами формы, путем нажатия на кнопку, необходимо выполнить ряд действий.

Чтобы формы были видны одна из другой, надо в h-файл главной формы после *#pragma once* поместить операторы:

```
#include "Form2.h"
```

Затем необходимо написать обработчик события **Нажатие на кнопку**: дважды кликнуть по компоненту кнопка **Button1** (**ОТКРЫТЬ**) на первой форме, в открывшемся окне точно в месте мигания курсора написать [1]:

```
Form2 ^f2=gcnw Form2();
```

```
f2->Show();
```

для второй кнопки (**ВЫХОД**):

```
this->Close();
```

На второй форме для первой кнопки (**ВЫХОД 1**) обработчик

события:

```
this->Close();
```

для второй кнопки (**ВЫХОД 2**):

```
MessageBox::Show("Ошибка закрытия формы", "Сообщение");
```

Внесите изменения в свойства кнопок, используя окно **Свойства**: цвет кнопки(**BackColor**), надпись кнопки (**Text**), цвет и шрифт надписи на кнопке (**Font**), другой вид курсора(**Curcor**), размер кнопки(**Size**).

Ознакомьтесь с основными свойствами и событиями формы, читая их характеристики при перемещении курсора в поле соответствующего свойства или события в окне свойств.

Запустите проект на выполнение и проверьте его работу.

В данном случае в обработчике события для открытия второй формы используется код:

```
Form2 ^f2=gcnnew Form2();
```

```
f2->Show();
```

Данная запись объясняется тем, что C++/CLI является своего рода адаптацией языка C++ к среде программирования .NET. При этом C++ стандарта ISO интегрируется в Общую языковую инфраструктуру (Common Language Infrastructure, CLI). Как уже говорилось, в CLR используется «автоматическая сборка мусора», в результате которой в процессе дефрагментации после удаления неиспользуемых объектов адреса элементов могут изменяться. Для решения данной проблемы в C++/CLI применяется механизм доступа к объектам в куче с обновлением адресов. Одним из способов реализации данного механизма являются отслеживаемые дескрипторы, которые в отличие от указателей в C++ хранят адрес, обновляемый при перемещении объекта во время «сборки мусора». Для размещения объектов в куче используется **gcnew**, а спецификатором дескриптора является символ **^**.

Для вызова формы в данном случае используется метод **Show()**, позволяющий пользователю переключаться между двумя открытыми формами. Также можно использовать и метод **ShowDialog()**, но тогда при открытой второй форме нельзя будет обратиться к первой и продолжить работу приложения. Такой режим называется модальным.

Наиболее часто используемыми методами формы наряду с **Show()** и **ShowDialog()** являются следующие:

- **Close()** – форма закрывается. Причем если она главная, то закрывается и приложение.
- **Hide()** – форма невидима.
- **Focus()** – форма активна, то есть видима и доступна.

2.7. Свойства формы

Свойства формы встречаются у многих компонентов. Более детально с ними можно познакомиться, вызвав справку путем выделения формы или другого компонента и нажав клавишу <F1>.

Воспользовавшись окном Свойства можно получить краткую информацию о каждом свойстве внизу окна. Если выделить несколько объектов, то можно одновременно изменить их свойства. Так как свойства подробно описаны во многих изданиях ограничимся рассмотрением только тех из них, которые необходимы для решения конкретной практической задачи.

Рассмотрим некоторые свойства формы, определяющие следующие ее характеристики:

- **AcceptButton** – позволяет закрепить за клавишей <Enter> действия обработчика события любой кнопки формы из выпадающего списка.
- **AutoScaleMode** – обеспечивает адаптацию размера формы и компонентов формы к изменениям в зависимости от воздействия, определяемого выбранным режимом: **Font** – размер шрифта операционной системы, **DPI** – размер экрана, **Inherit** – использование шрифта и разрешения базового компьютера на другом компьютере.
- **AutoScroll** – использование полос прокрутки.
- **AutoSize** – изменение размера элемента в соответствии с содержимым.
- **AutoSizeMode** – режим изменения размеров формы: **GrowAndShrink** – может увеличиваться и уменьшаться, **GrowOnly** – только увеличиваться.
- **BackColor** – цвет фона.
- **BackgroundImage** – фоновое изображение.

– **BackgroundImageLayout** – как размещается фоновое изображение: **None** – выравнивается вверх и влево, **Tile** – мозаикой, **Center** – по центру, **Stretch** – растягивается на весь компонент без соблюдения пропорций, **Zoom** – как и **Stretch**, но с соблюдением пропорций.

– **CancelButton** – позволяет закрепить за клавишей <Esc> действия обработчика события любой кнопки формы из выпадающего списка.

– **ContextMenuStrip** – подключает к компоненту любое контекстное меню, определенное на форме и выбранное из выпадающего списка.

– **ControlBox** – отображение кнопок закрытия и изменения размера в заголовке.



– **Cursor** – вид курсора при его нахождении над компонентом.

– **Enabled** – возможность доступа к компоненту.

– **Font** – шрифт текста на компоненте.

– **ForeColor** – цвет текста на компоненте.

– **FormBorderStyle** – стиль границы и заголовка формы: **None** – граница и заголовок отсутствуют, **FixedSingle** – фиксированная граница (одна линия), **Fixed3D** – трехмерная фиксированная граница, **FixedDialog** – утолщенная фиксированная граница, **Sizable** – изменяемая граница, **FixedToolWindow** – фиксированная граница, в заголовке только кнопка закрытия формы, **SizableToolWindow** – изменяемая граница, в заголовке только кнопка закрытия формы.

– **MaximizeBox**, **MinimizeBox** – кнопка свертывания и разворачивания формы.

– **HelpButton** – кнопка справки в заголовке, видна, только если скрытаны кнопки свертывания и разворачивания, действие этой кнопки определяется в обработчике события формы **HelpRequested**, который выполняется также и при нажатии клавиши <F1>.

– **Icon** – значок, отображаемый в строке заголовка формы и на панели задач при запуске приложения.

– **Opacity** – прозрачность формы, увеличивается с уменьшением процентов.

– **Padding** – отступы от границы до содержимого компонента.

– **Size** – размеры компонента.

– **SizeGripStyle** – калибровочная полоска в правом нижнем углу формы.

– **Location** – координаты левого верхнего угла формы или другого компонента.

– **StartPosition** – начальное положение формы при выполнении приложения: **CenterScreen** – в центре экрана, **Manual** – устанавливается в **Location**.

– **Text** – текст, отображаемый на компоненте, для формы – в ее заголовке.

– **TopMost** – расположение формы поверх других.

– **WindowState** – начальное состояние формы при запуске приложения: **Normal** – размеры, установленные в конструкторе, **Minimized** – отображается в свернутом виде (кнопка на панели задач), **Maximized** – развернута на весь экран.

Часто используемые события формы вызываются следующими действиями:

Activated – форма активируется.

Click – щелчок мышью по форме.

HelpButtonClicked – нажатие кнопки Справка в строке заголовка формы.

HelpRequested – нажатие клавиши F1 или кнопки контекстной справки.

Load – каждая загрузка формы.

Shown – первое отображение формы.

Справку с примером использования для каждого события можно вызвать также как и справку о компоненте, нажав клавишу <F1>, когда курсор находится в соответствующей строке.

Глава 3.

Исследование базовых компонентов интерфейса приложения

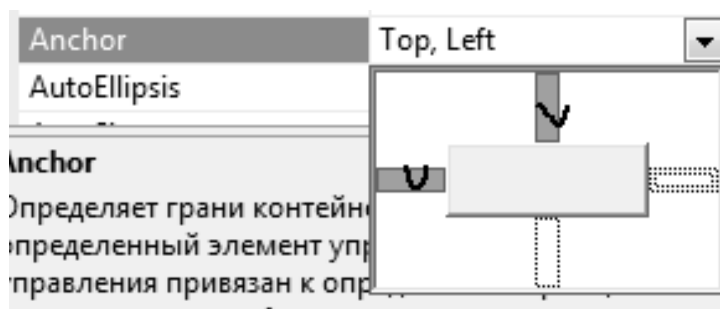
3.1. Компоненты Button, Panel, Label, TextBox, PictureBox

Для разработки первых приложений мы будем использовать компоненты, наиболее часто используемые при создании интерфейса: кнопки, панели, надписи, текстовые поля и изображения. Все перечисленные компоненты, кроме компонента **Panel**, располагаются в окне **Панель элементов** в группе **Стандартные элементы управления**. Рассмотрим их основные свойства, события и методы.

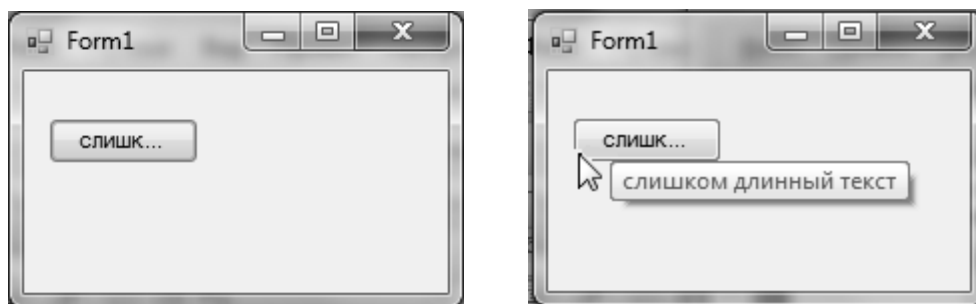
Компонент Button

Компонент **Button** представляет собой кнопку, для которой характерны некоторые события, свойства и методы, как и для формы. Однако имеются и отличающие данный компонент свойства:

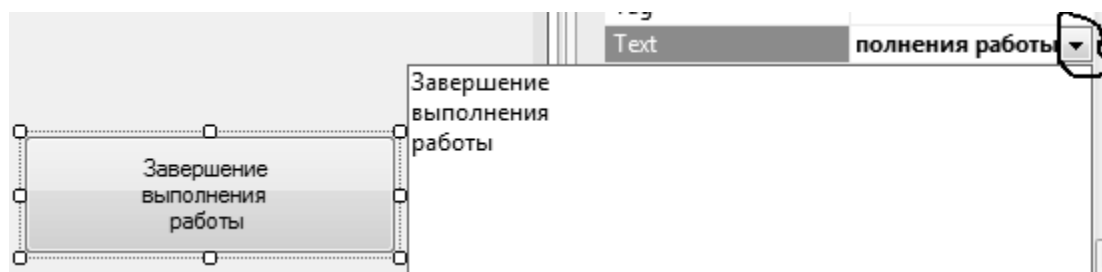
– **Anchor** – определяет, к какой грани контейнера привязан компонент. При изменении размера формы расстояние от выбранных граней не меняется. На рисунке выбрана привязка к левому и верхнему краю формы. При этом **AutoSize** должно быть **false**.



– **AutoEllipsis** – если длина текста превосходит ширину кнопки, то при выборе этого свойства вместо непоместившегося текста отобразится многоточие, а при наведении курсора он будет виден во всплывающей подсказке. При этом **AutoSize** должно быть **false**.



– **Text** – текст на элементе. Используя кнопку в левом крае поля, можно открыть область для ввода текста и расположить текст в несколько строк, даже если ширина компонента позволяет однострочное изображение.

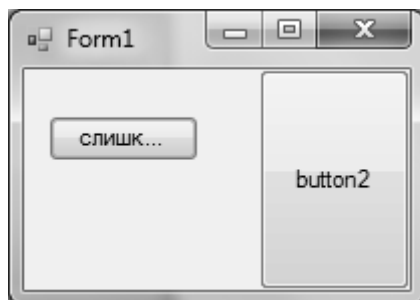


– **DialogResult** – используется только для кнопок на формах, открытых как модальные(с помощью метода **ShowDialog()**). При выборе любого значения этого свойства кроме **None** нажатие кнопки приведет к тому, что форма закроется без использования какого-либо обработчика события, а свойству формы **DialogResult** присвоится то значение, которое мы выбрали в одноименном свойстве кнопки: **Yes**, **No**, **Cancel**, **OK**, **Ignore**, **Abort** или **Retry**.

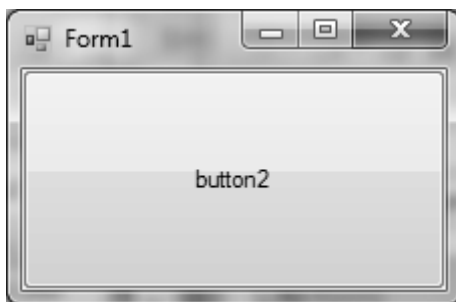
– **Dock** – определяет, к какой стороне формы будет присоединен компонент. При выборе варианта, указанного на рисунке



кнопка располагается вдоль правой границы формы.

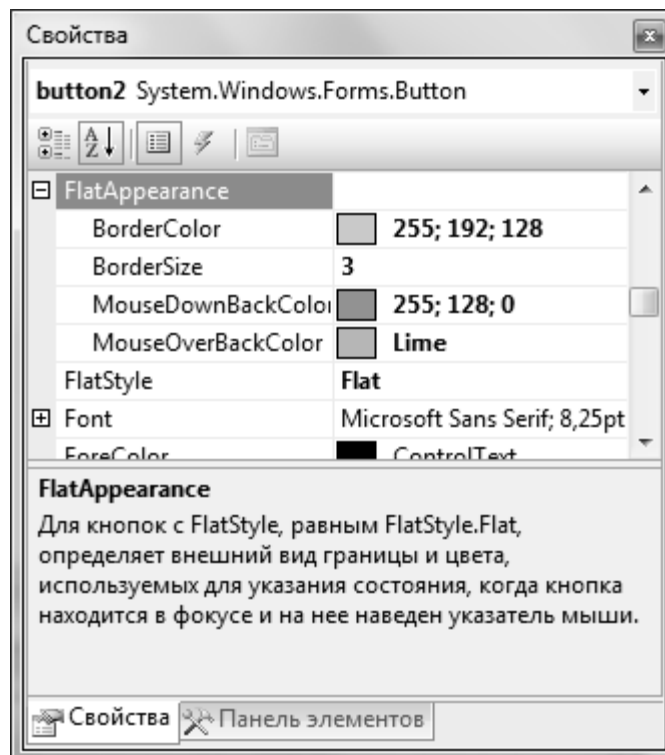


Если выбрать центральный прямоугольник (**Fill**), кнопка растянется на всю форму.

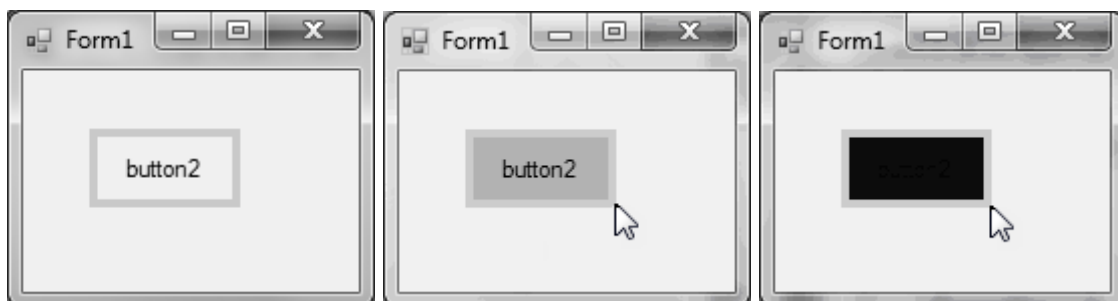


– **FlatStyle** – внешний вид элемента: **Flat** – плоский, **Popup** – плоский, при наведении курсора – объемный, **Standard** – объемный, **System** – определяется операционной системой.

– **FlatAppearance** – определяет ряд характеристик кнопки:
цвет границы (**BorderColor**),
ширина границы (**BorderSize**),
цвет кнопки при наведении на нее курсора(**MouseOverBackColor**),
цвет кнопки при нажатии на кнопку (**MouseDownBackColor**).
при этом свойство **FlatStyle** должно иметь значение **Flat**.



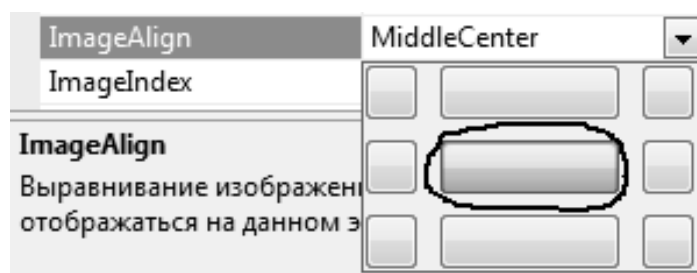
При выборе соответствующих значений внешний вид кнопки будет меняться в процессе выполнения приложения в зависимости от выполняемых с ней действий.



– **Image** – изображение на элементе (выбрать нажав на кнопку с многоточием). **FlatStyle** должно быть **System**.

– **ImageAlign** – выравнивание изображения на элементе.

При выборе варианта как на рисунке, изображение располагается в центре кнопки.



– **TabStop** – перемещение между элементами с помощью клавиши **<Tab>**.

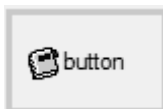
– **TabIndex** – порядковый номер элемента при перемещении между элементами с помощью клавиши **<Tab>**.

– **TextImageRelation** – варианты расположения изображения и текста на кнопке:

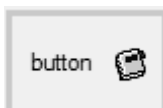
Overlay – изображение и текст в одном месте



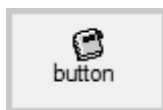
ImageBeforeText – изображение перед текстом горизонтально



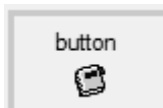
TextBeforeImage – текст перед изображением горизонтально



ImageAboveText – изображение над текстом



TextAboveImage – текст над изображением



– **UseMnemonic** – в качестве мнемонической клавиши определяется та, перед которой в свойстве **Text** записан символ **&**. Например, **&button**. Теперь при работе приложения нажатие клавиши **b** вызовет обработчик события нажатия на данную кнопку.

- **UseVisualStyleBackColor** – визуальные стили фона.
- **Visible** – видимость элемента.

Информацию о событиях компонента **Button** можно прочитать в нижней части окна **Свойства** при выборе соответствующего события. Чаще всего используются следующие события:

- **Click** – щелчок по кнопке.
- **Enter** – получение фокуса ввода (кнопка активна).
- **MouseHover** – задержка курсора над кнопкой.
- **MouseLeave** – курсор покидает область кнопки.

Некоторые методы **Button**:

- **Hide ()** – спрятать кнопку.
- **Show()** – показать кнопку.
- **Focus ()**, **Select ()** – выделить кнопку(сделать активной).

Компонент **Panel**

Данный компонент располагается в окне **Панель элементов** в группе **Контейнеры**. На **Panel**, как и на форму можно добавлять другие компоненты, которые объединяются и при редактировании перемещаются при перемещении панели. Чтобы были видны все элементы, у панели регулируются свойства, управляющие параметрами полос прокрутки: **AutoScroll**, **AutoSize** и **AutoSizeMode**, имеющие то же назначение, что и у формы. События **Panel** имеют сходное значение с событиями формы.

Компонент **Label**

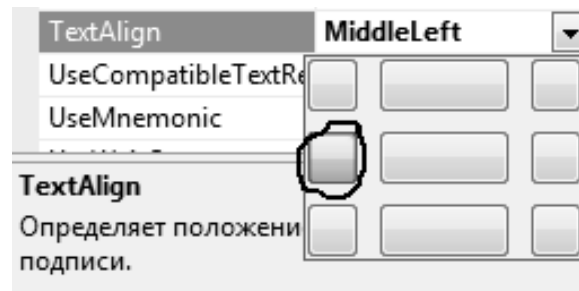
Компонент **Label** предназначен для отображения различных надписей на форме, которые не могут изменяться пользователем при работе приложения. Как правило, **Label** используется для поясняющих подписей к другим элементам формы, вывода сообщений, результатов.

Основные свойства и события **Label** те же, что и у рассмотренных ранее компонентов.

После размещения на форме размеры **Label** устанавливаются автоматически, адаптируясь к размерам текста на элементе, то есть свойство **AutoSize** имеет значение **true**. Если нужно изменить разме-

ры компонента принудительно с помощью маркеров по углам и сторонам, или, установив другие значения в свойстве **Size**, то **AutoSize** необходимо установить значение **false**.

При этом же условии будет доступно для изменения и свойство **TextAlign**, которое позволяет выбрать вариант выравнивания текста, размещенного на элементе. Для этого нужно воспользоваться выпадающим списком в поле значения и щелчком выбрать подходящий вариант.

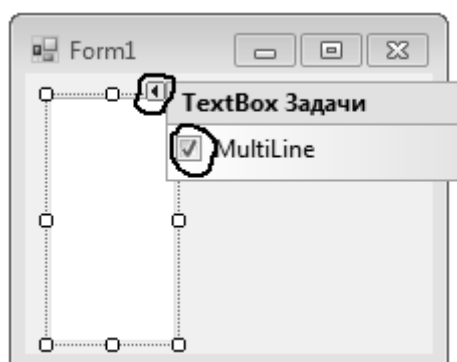


Свойство **BorderStyle** позволяет выбрать тип границы элемента: **None** – невидимая, **Fixed3D** – трехмерная, **FixedSingle** – одна линия.

Компонент **TextBox**

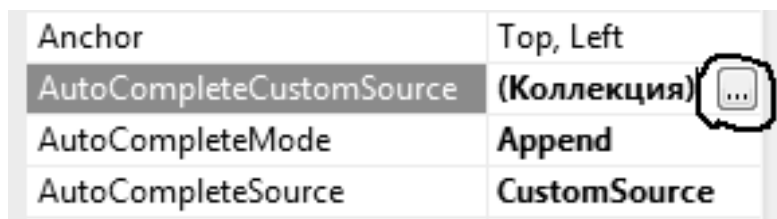
Компонент **TextBox** служит для ввода и вывода текстовой информации, редактирования и отображения многострочного текста, ввода пароля и т.д.

Компонент может использоваться как в однострочном (по умолчанию), так и в многострочном режиме. Переход между режимами осуществляется путем установки соответствующего значения свойству **MultiLine** в окне **Свойства** или непосредственно на компоненте с помощью флажка.

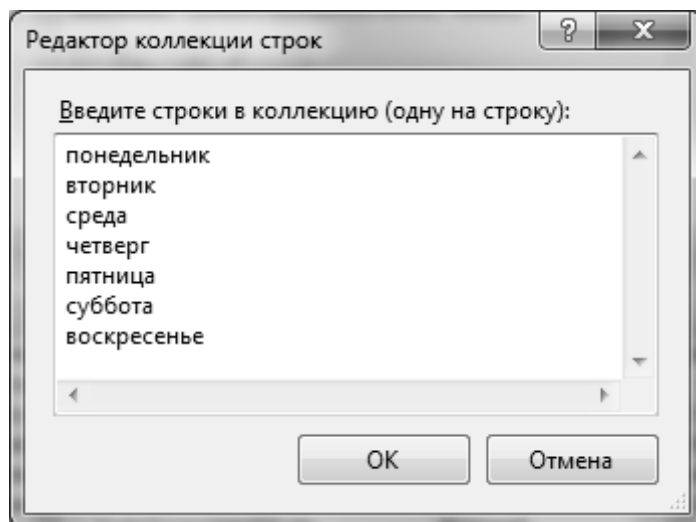


При использовании данного компонента в однострочном режиме можно обеспечить контроль ввода в виде автоматической подсказки для заполнения, которая появляется после ввода первой буквы слова. Источник данной подсказки определяется в свойстве **AutoCompleteSource** и может быть следующих видов: **AllSystemSources** – файловая система и URL-адреса, **AllUrl** – все URL-адреса, **FileSystem** – файловая система (имена файлов и папок появляются после набора наклонной черты \), **HistoryList** – URL-адреса из журнала, **RecentlyUsedList** – последние использованные URL-адреса, **CustomSource** – строки из **AutoCompleteStringCollection**, **FileSystemDirectories** – имена каталогов (появляются после набора наклонной черты \ или имени диска, например, **c:**) и **None** – источника нет.

Если в качестве значения свойства **AutoCompleteSource** выбирается **CustomSource**, то для обеспечения его работы надо использовать свойство компонента **AutoCompleteCustomSource**.



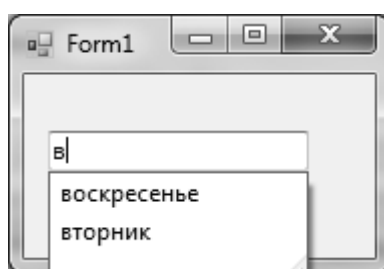
Нажав кнопку с многоточием в поле данного свойства, можно открыть редактор для ввода строк вариантов автоматического заполнения.



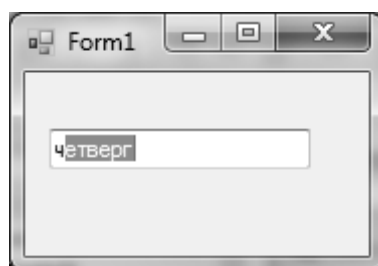
Кроме того для обеспечения работы данной функции нужно выбрать нужный вариант отображения текста подсказки, который задается в свойстве **AutoCompleteMode**. По умолчанию значение данного свойства **None**, то есть подсказка не отображается.

Тогда в работающем приложении достаточно ввести в **TextBox** одну или несколько букв из начала слова и нажать ввод, чтобы добавить нужный текст. Свойство **AutoCompleteMode** может принимать также следующие значения, характеризующие вид отображения предложенных строк:

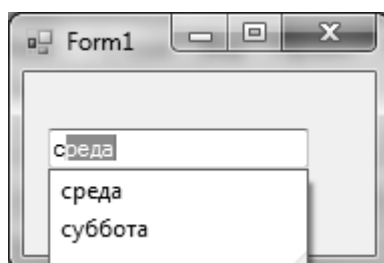
Suggest – вспомогательный раскрывающийся список,



Append – выделенный текст в поле как продолжение уже введенных букв,



SuggestAppend – сочетание двух вышеописанных вариантов.

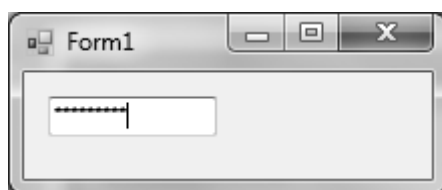


Рассмотрим некоторые свойства **TextBox**, отличные от уже знакомых.

– **HideSelection** – текст сохраняет выделение при переключении фокуса на другой компонент, если выбрать **false**.

– **Lines** – текст в виде массива строк можно редактировать, нажав кнопку с многоточием в поле данного свойства.

– **PasswordChar** – символ, который заменяет вводимый пароль. Доступно в однострочном режиме.



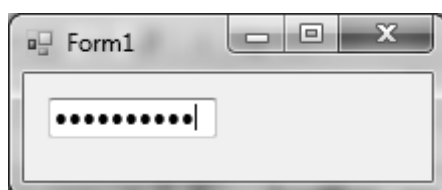
– **ReadOnly** – текст нельзя редактировать, если значение свойства **true**.

– **ScrollBars** – полосы прокрутки в многострочном режиме: **None** – отсутствуют, **Horizontal** – горизонтальные, **Vertical** – вертикальные, **Both** – обе.

– **ShortcutsEnabled** – активность «горячих клавиш».

– **TextAlign** – выравнивание текста.

– **UseSystemPasswordChar** – текст при вводе заменяется символами пароля. При этом нельзя использовать свойство **PasswordChar**.



– **WordWrap** – автоматический перенос слов на следующую строку, если длина текста превышает ширину компонента. Используется в многострочном режиме.

События **TextBox** во многом сходны с событиями других компонентов. Более подробно можно ознакомиться с их назначением в справочной системе, а краткая характеристика доступна внизу окна **Свойства**.

Часто в программировании применяется такой прием, как проверка ввода данных. Для этой цели можно использовать обработчик события **KeyDown**, которое происходит при нажатии какой-либо клавиши в момент, когда компонент **TextBox** находится в фокусе.

К примеру, мы хотим, чтобы при окончании ввода данных нажатием клавиши **<Enter>** выводилось сообщение «Ввод завершен», то мы должны составить соответствующий обработчик события. Для этого выделим на форме компонент **TextBox**, в окне **Свойства** перейдем на вкладку **События**, выполним двойной щелчок левой кнопкой мыши в поле события **KeyDown**. В открывшемся окне кода пишем в теле автоматически созданного обработчика события строго в точке нахождения курсора выделенный текст:

```
private: System::Void
textBox1_KeyDown(System::Object^ sender,
System::Windows::Forms::KeyEventArgs^ e) {

if(e->KeyCode == Keys::Enter)
MessageBox::Show("Ввод завершен") ;
}
```

В строке заголовка обработчика события **KeyDown** имеется **e** как параметр класса **KeyEventArgs** пространства имен **System:: Windows::Forms**, который определяет данные для событий «нажатие клавиши» и «отпускание клавиши». **KeyCode** – код клавиатуры для данных событий. **Keys** – определяет коды клавиш. Назначение вышеописанных составляющих можно узнать из всплывающих подсказок при наведении курсора на соответствующее слово. Таким образом, запись можно пояснить: если код клавиатуры при событии «нажатие клавиши» равен коду клавиши **<Enter>**, то на экран выведется сообщение «Ввод завершен».

Используя суфлер кода, который появится после набора **Keys::**, можно выбрать код любой другой клавиши.

Для **TextBox** наиболее актуальны методы для работы с текстом:

AppendText("текст") – добавить текст.

Clear() – очистить.

Copy() – копировать.

Cut() – вырезать.

Paste() – вставить.

Select() – выделить.

SelectAll() – выделить все.

DeselectAll() – снять выделение.

Focus() – установить фокус.

Undo() – отмена последнего действия.

Компонент **PictureBox**

Компонент предназначен для вывода изображения.

– **Image** – отображаемое изображение (**.jpeg**, **.gif**, **.png**) выбирается в диалоговом окне, открываемом после нажатия кнопки с многоточием в поле свойства.

– **ErrorImage** – изображение при ошибке загрузки картинки, указанной в свойстве **Image**.

– **InitialImage** – изображение на время загрузки требуемого.

– **SizeMode** – вариант размещения изображения на компоненте:

Normal – левый верхний угол изображения и компонента совпадают, размер изображения не меняется, отображается только то, что поместилось в **PictureBox** (для наглядности свойство **BorderStyle=FixedSingle**).



StretchImage – изображение вписывается в компонент без сохранения пропорций.



AutoSize – компонент принимает размеры изображения.



CenterImage – центр изображения и компонента совпадают, размер изображения не меняется, отображается только то, что поместилось в **PictureBox**.



Zoom – изображение вписывается в компонент с сохранением пропорций (для наглядности свойство `BorderStyle=FixedSingle`).



– **ImageLocation** – имя файла с указанием пути с изображением на диске. Например, `d:\\user\\1.jpg`. Если указать значение этого свойства, то при загрузке изображения с помощью метода **Load()** не нужно указывать в скобках, где находится изображение.

```
pictureBox2->Load();
```

Если свойство не определено, то надо указать путь в коде:

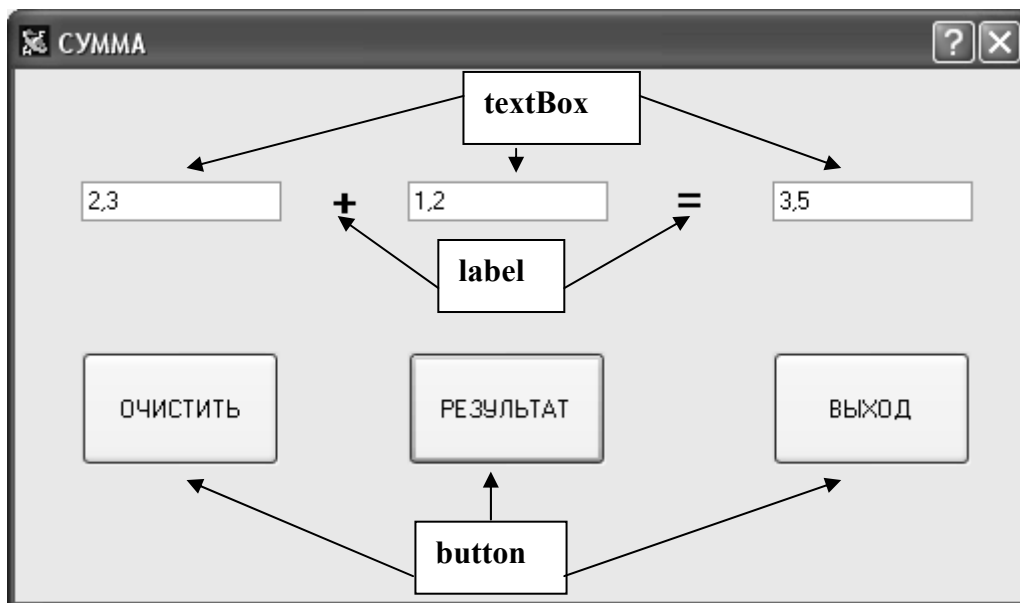
```
pictureBox2->Load("d:\\user\\1.jpg");
```

Задание для выполнения 1.

Разработать приложение «**summa**», которое выполняет сложение двух чисел, вводимых с клавиатуры, обеспечивает вывод результата на экран, очистку полей ввода-вывода, получение справки о назначении разработанного приложения, выход из приложения по нажатию на соответствующую кнопку и с помощью клавиши **<Esc>**.

Для получения результата нужно выполнить следующие шаги:

1. Создать проект.
2. Добавить на форму компоненты согласно рисунку.



3. У формы изменить свойства:

Text=СУММА, **MaximizeBox=False**, **MinimizeBox=False**, **HelpButton=True**, **Icon** (выбрать файл с именем *.ico), **CancelButton**(выбрать в выпадающем списке свойства имя кнопки с обработчиком события, выполняющим закрытие приложения – **ВЫХОД**).

4. Обработчик события кнопки **ОЧИСТИТЬ**:

```
textBox1->Clear();
textBox2->Clear();
textBox3->Clear();
```

5. Обработчик события кнопки **РЕЗУЛЬТАТ**:

```
float xr, yr, zr;
xr=Convert::ToSingle(textBox1->Text);
yr=Convert::ToSingle(textBox2->Text);
zr=xr+yr;
textBox3->Text=zr.ToString();
```

6. Обработчик события кнопки **ВЫХОД**:

```
Close();
```

7. Обработчик события формы **HelpRequested** (для отображения справки):

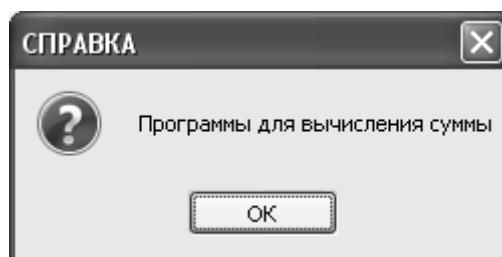
```
MessageBox::Show("Программы для вычисления  
суммы", "СПРАВКА", MessageBoxButtons::OK, Message  
BoxIcon::Question);
```

Формирование окна вывода сообщения с помощью функции `MessageBox::Show()` состоит из следующих частей:

```
// "Программы для вычисления суммы" – выводится  
внутри окна, если нужно вывести текст, то его берут в кавычки;  
// "СПРАВКА" – текст в заголовке окна;  
// MessageBoxButtons::OK – вид кнопки в окне (OK в  
данном случае);  
// MessageBoxIcon::Question – вид изображения внутри  
окна (Question в данном случае).
```

При записи данной функции вас будут сопровождать пояснения в виде всплывающей подсказке о назначении каждой части.

В итоге в работающем приложении при выборе кнопки справка в заголовке окна формы и щелчке по любому компоненту появится следующее окно сообщения



3.2. Использование переменных различных типов

В предыдущем примере мы уже использовали переменные типа **float** и выполняли их преобразование для вывода в текстовое поле. Типы данных для приложений CLR определены в пространстве имен **System**. При этом переменные основных типов данных в C++/ CLI мо-

гут рассматриваться и в качестве значения, и в качестве объекта, обладая рядом дополнительных свойств. Пользователь выбирает варианты использования таких переменных в зависимости от решаемой задачи.

Каждому основному типу данных соответствует класс значений **CLI**. Более подробную информацию о типах данных можно получить из справочной системы. Здесь рассмотрим только те, которые будут использованы нами при рассмотрении практических примеров.

Логический тип данных **bool** соответствует классу **System::Boolean**, целые числа типа **int** – классу **System::Int32**, вещественные **float** – классу **System::Single**, строка символов **Unicode String** – классу **System::String**.

Если мы используем объект класса ссылочного типа, то переменные, ссылающиеся на них должны быть отслеживаемыми дескрипторами.

Определим переменную как отслеживаемый дескриптор типа **String**.

```
String^ s1;
```

Такую переменную можно инициализировать при объявлении:

```
String^ s1="text";
```

Теперь в динамической памяти имеется объект класса **String**, получивший значение **"text"**.

Таким же образом можно использовать дескрипторы и в случае других типов, например, **float^ a**. Теперь, **a** – это не переменная, а объект, расположенный в динамической памяти с автоматической очисткой. Чтобы его использовать, например, в операции сложения необходимо использовать операцию разадресации *****:

```
y=*a+2;
```

по аналогии с обычными указателями для использования значения, хранящегося по адресу в дескрипторе.

Для представления объекта любого класса **C++/CLI** в виде строки используется функция **ToString()**. Причем строковое представление можно получить, даже не вызывая явно данную функцию.

Некоторые действия со строками можно выполнить различными способами. Так объединить строки можно и с помощью функции

Concat() и с помощью операции сложения:

```
//объявление строк
String ^mn="Вас", ^nm="Приветствую ",
^b="ЗДРАВСТВУЙТЕ!\n";
b=b->Concat(nm,mn); //объединение строк
label1->Text=b; //вывод строки b в надпись label1
```

Надпись на экране будет выглядеть так: **Приветствую Вас**

Тот же результат получится, если объединение строк выполнить следующим образом:

```
b=nm+mn;
```

Строковой переменной можно присвоить значение свойства **Text** компонента **textBox** или другого компонента:

```
nm=textBox1->Text;
```

Со строками можно выполнять многие действия. К примеру:

```
- очистить    b=b->Empty;
- копировать  b=b->Copy(nm);
- определить длину  int n=b->Length;
- преобразовать число в строку
label1->Text=n.ToString();
```

или

```
label1->Text=Convert::ToString(n);
```

- преобразовать строку в целое число

```
String ^mn="123";
int n=n.Parse(mn);
```

или

```
int n=Convert::ToInt32(mn);
```

- преобразовать строку в вещественное число

```
String ^mn="123,5";
```

```
float n=Convert::ToDouble(mn);
```

– преобразовать строку в формат DateTime

```
String ^m="03/03/2007";
```

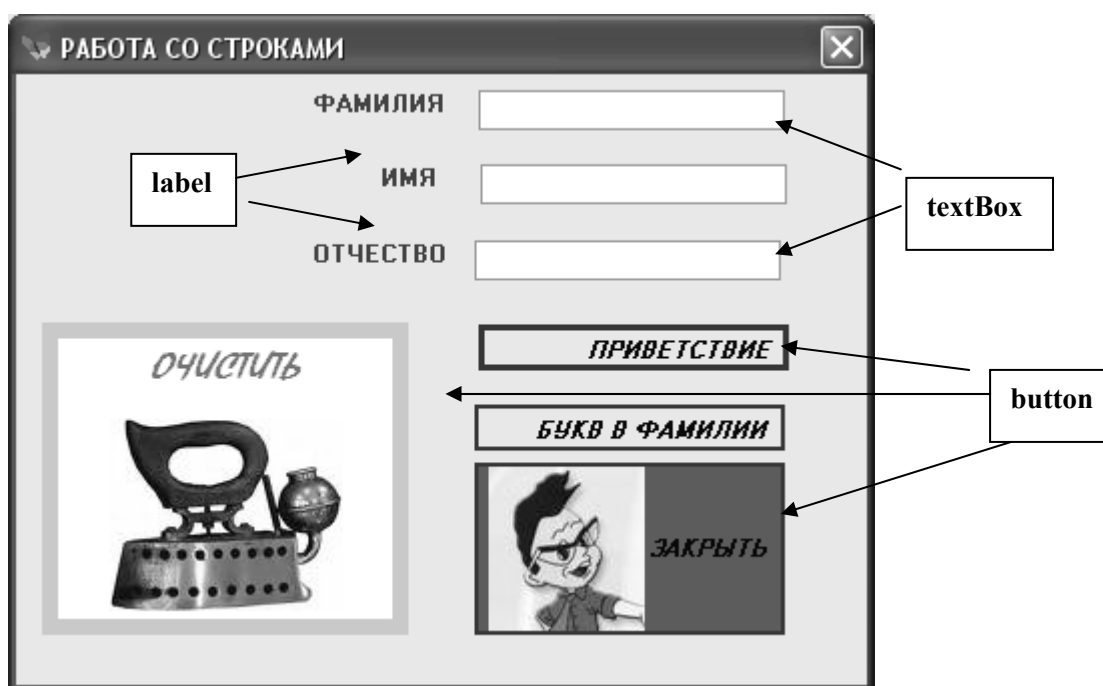
```
DateTime d = Convert::ToDateTime(m);
```

Как видно, везде используется `Convert::To+ИмяТипа`.

Задание для выполнения 2.

Разработать приложение «*stroki*», обеспечивающее ввод в соответствующие поля фамилии, имени и отчества, очистку данных полей, подсчет букв в фамилии, вывод приветствия в окне сообщения и закрытие приложения. Для получения результата нужно выполнить следующие шаги:

1. Создать проект.
2. Добавить на форму компоненты согласно рисунку.



3. Обработчик события «Нажатие на кнопку ПРИВЕТСТВИЕ»:

```
String^ fp, ^ip, ^op, ^pp="ЗДРАВСТВУЙТЕ, ",  
        ^xp="\n РАДЫ ПРИВЕТСТВОВАТЬ ВАС!!!",
```

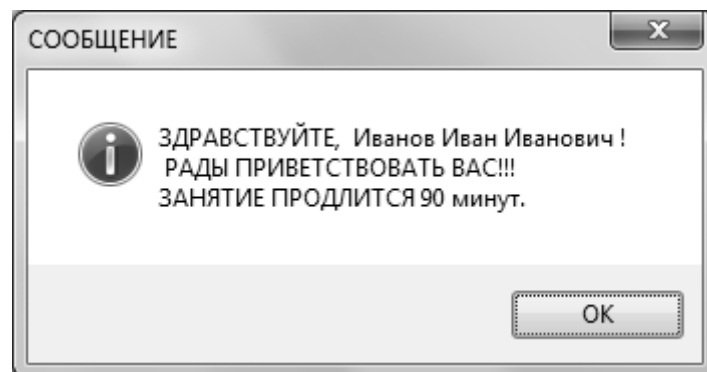
```

        ^np="\nЗАНЯТИЕ ПРОДЛИТСЯ ",
^mp="минут.";
        int gp=90;
        fp=textBox1->Text+" ";
        ip=textBox2->Text+" ";
        op=textBox3->Text+" ";

        MessageBox::Show(pp+fp+ip+op+"!" +xp+np+gp+mp,
        "СООБЩЕНИЕ", MessageBoxButtons::OK,
        MessageBoxIcon::Information);

```

В результате после ввода в соответствующие поля фамилии, имени и отчества (например, Иванов Иван Иванович) и нажатия на кнопку появится:



4. Обработчик события «Нажатие на кнопку **БУКВ В ФАМИЛИИ**»:

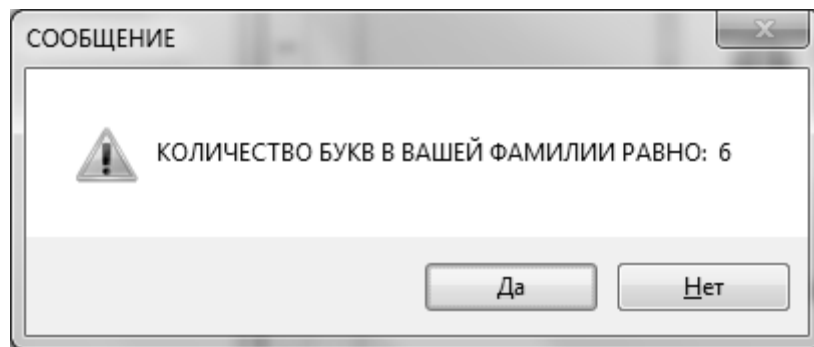
```

String^ mn, ^nm, ^b="ЗДРАВСТВУЙТЕ!\n", ^x;
        mn=textBox1->Text;
        int i=mn->Length;

        MessageBox::Show("КОЛИЧЕСТВО БУКВ В ВАШЕЙ ФАМИЛИИ РАВНО:" +i, "СООБЩЕНИЕ", MessageBoxButtons::YesNo,
        MessageBoxIcon::Exclamation);

```

Результат на экране:



5. У кнопок **ЗАКРЫТЬ** и **ОЧИСТИТЬ** установить свойства:

FlatStyle=Flat, **FlatAppearance** и **Image** (по своему вкусу),
TextAlign и **ImageAlign** (в соответствии с картинкой).

3.3. Разработка приложений с использованием классов

Задание для выполнения 3.

Разработать приложение «**Калькулятор**», используя класс **calc**.

Для получения результата нужно выполнить следующие шаги:

1. Создать проект.
2. Добавить на форму компоненты согласно рисунку

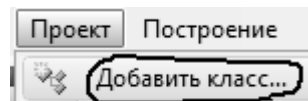


3. Создать класс **calk**, содержащий:

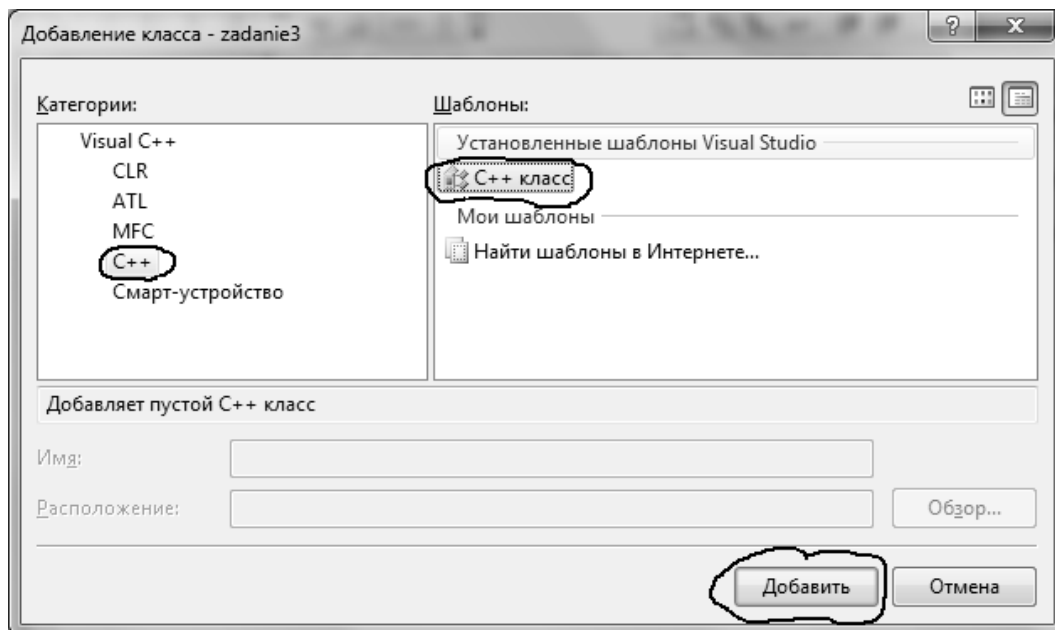
- две закрытые переменные вещественного типа **a** и **b**;
- конструктор для инициализации данных переменных;
- открытые функции сложения, вычитания, умножения, деления

Чтобы создать класс в Microsoft Visual Studio 2008 необходимо выполнить команды:

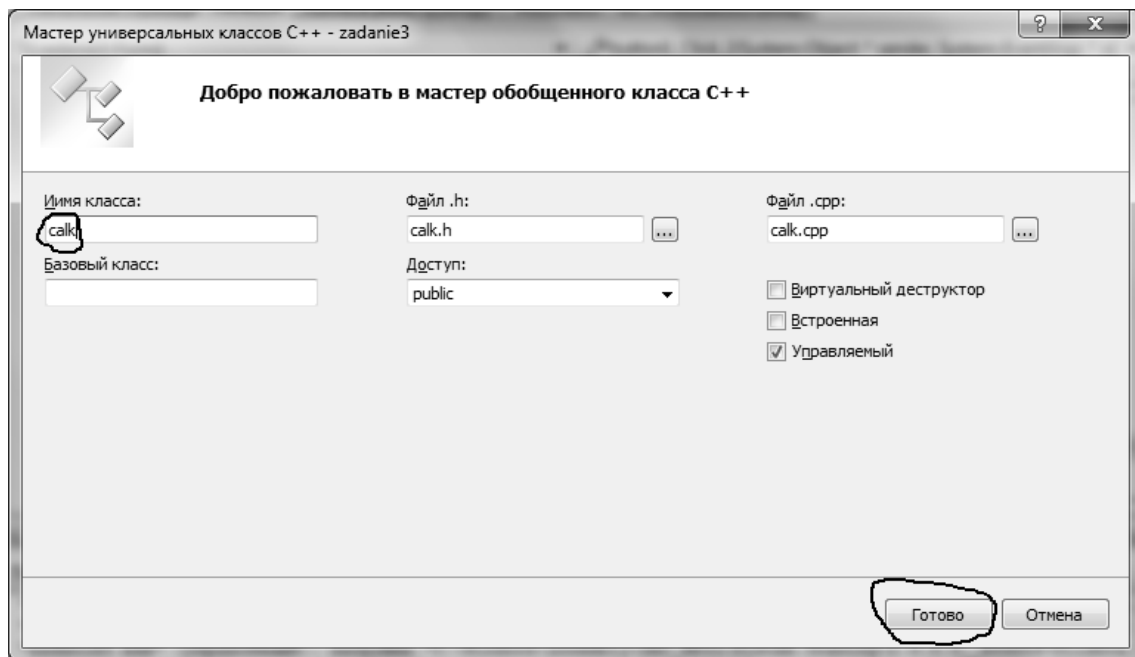
- главное меню: **Проект/Добавить класс.../**



- выбрать категорию **C++/ шаблон C++ класс/**
- нажать кнопку **Добавить**



- ввести в соответствующее поле **Имя класса** (имена файлов с расширением .h и .cpp пропишутся сами) и нажать кнопку **Готово**.



4. Изменить содержимое открывающегося файла `calk.h` :

```
#pragma once
ref class calk
{float a,b;
public:
    calk(float xc,float yc); //конструктор с
параметрами
    float sum();
    float razn();
    float umn();
    float del();
};
```

5. Изменить содержимое файла `calk.cpp`:

```
#include "StdAfx.h"
#include "calk.h"

calk::calk(float xc,float yc)
{
    a=xc;
    b=yc; }
}
```

```
float calk::sum()  
{return (a+b);}
```

```
float calk::razn()  
{return a-b;}
```

```
float calk::umn()  
{return a*b;}
```

```
float calk::del()  
{return a/b;}
```

6. Изменить содержимое файла Form1.h:

```
#pragma once      //это не меняем!!!!  
#include "calk.h"  //подключаем заголовочный файл  
класса
```

```
namespace калькулятор {    //это не меняем!!!!  
//калькулятор -это имя моего приложения, у вас  
//будет такое имя какое укажете при создании  
//проекта
```

```
//определяем глобальные переменные  
float operator1, operator2;  
int operation;
```

```
using namespace System; //это не меняем!!!!  
using namespace System::ComponentModel;  
using namespace System::Collections;  
using namespace System::Windows::Forms;  
using namespace System::Data;  
using namespace System::Drawing;
```

7. Выполнение события «**Нажатие на кнопку**» (**Click**) для кнопок с цифрами должно приводить к появлению цифры, указанной на кнопке в текстовом поле **textBox1**.

Для каждой кнопки надо прописать свой обработчик события:

```
//для кнопки с цифрой 1
textBox1->Text=textBox1->Text+"1";
//для кнопки с цифрой 2
textBox1->Text=textBox1->Text+"2";
```

и так далее.

8. Окно справки в виде текстового сообщения должно быть различным для текстового поля и всей формы (обработчик события **HelpRequested** для **textBox1** и для **Form1**) – использовать `MessageBox::Show`.

9. Обработчики событий кнопок с изображением знака операции устанавливают значения глобальных переменных **operation** (число, соответствующее знаку операции **1** – сумма, **2**- вычитание, **3** – умножение, **4** - деление) и **operator1** (первое введенное в текстовое поле число). Также проверяется условие заполненности текстового поля значением числа. После данных операций выполняется очистка текстового поля. Для каждой кнопки с изображением знака операции (+, −, *, /) надо прописать свой обработчик события:

```
//обработчик для кнопки со знаком +
operation=1;
if (textBox1->Text!="")
operator1=Convert::ToSingle(textBox1->Text);
else operator1=0;
textBox1->Clear();

// обработчик для кнопки со знаком -
operation=2;
if (textBox1->Text!="")
operator1=Convert::ToSingle(textBox1->Text);
else operator1=0;
textBox1->Clear();
```

10. В обработчике события «**Нажатие на кнопку со знаком =**» устанавливается значение второго значения вводимого в текстовое поле, участвующего в вычислении (**operator2**). Также здесь объявляется с одновременной инициализацией объект класса **calk**. Обращение к функциям класса выполняется в зависимости от выбранной операции. Для этой цели используется оператор **switch()**. Полученный результат выводится в текстовое поле:

```
float rez;  
operator2=Convert::ToSingle(textBox1->Text);  
calk s(operator1,operator2);  
    switch(operation)  
    {  
        case 1: rez=s.sum();break;  
        case 2: rez=s.razn();break;  
        case 3: rez=s.umn();break;  
        case 4: rez=s.del();break;  
    }  
textBox1->Text=rez.ToString();
```

Задание для выполнения 4.

Разработайте приложение «**Угол**», используя класс **Angle**.

Для получения результата нужно выполнить следующие шаги:

1. Определите класс **Angle** (значение угла), в состав которого включена закрытая переменная **d** типа **double** и три открытых функции:

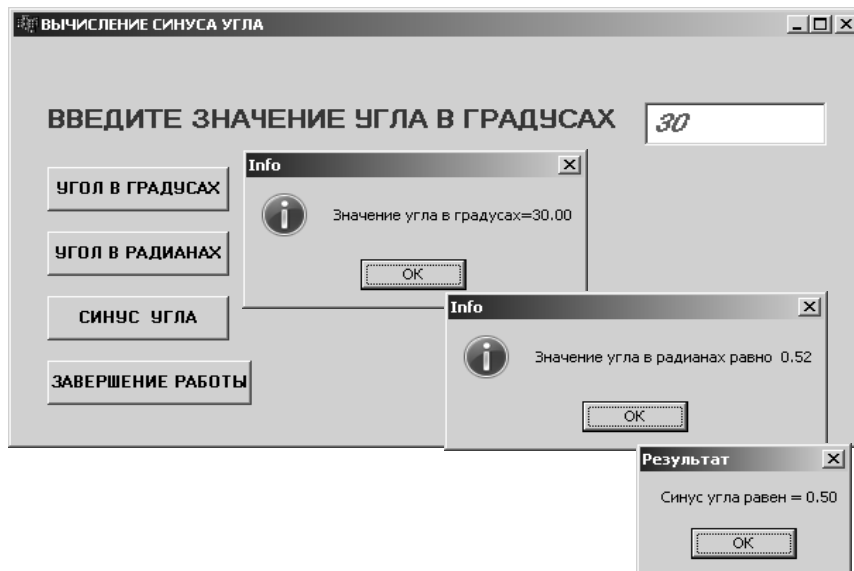
print_d() - для вывода на экран значения угла в градусах;

get_rad() - для перевода значения угла из градусов в радианы;

get_sinuso() - для вычисления синуса значения угла.

Для данного класса, определенного в файле **angle.h**, напишите программу, демонстрирующую перевод значения угла из градусов в радианы и вычисление синуса угла. Для вывода результатов используйте функцию **MessageBox**.

Рекомендуемый интерфейс приложения представлен на рисунке.



Создать класс так, как описано в предыдущем задании.

Для использования тригонометрических функций в файле определения функций класса **angle.cpp** необходимо использовать запись:

```
using namespace System;
```

Чтобы применить **MessageBox::Show()** записывается

```
using namespace System::Windows::Forms;
```

При обращении к тригонометрическим функциям применять следующую запись:

```
//перевод угла из градусов(d) в радианы (a)  
a=Math::PI*d/180.0;
```

```
//вычисление синуса угла с (угол в радианах)  
b=Math::Sin(c);
```

Файл angle.h

```
#pragma once
ref class angle
{
double d;
public:
    angle(double n); //конструктор

void print(double d); //объявление функций
double get_rad(double d);
double get_sinus(double d);
};
```

Файл angle.cpp

```
#include "StdAfx.h"
#include "angle.h"
    using namespace System;
    using namespace System::Windows::Forms;

angle::angle(double n)
{
d=n;
}
void angle::print(double d)
{
MessageBox::Show("Значение угла в градусах= "
+d,"Info",MessageBoxButtons::OK,MessageBoxIcon::
Information);
}

double angle::get_rad(double d)
{
double a=Math::PI*d/180.0;
return a;
}
```

```
double angle::get_sinus(double d)
{
double c,b;
c=angle::get_rad(d);
b=Math::Sin(c); //вычисление синуса угла c (угол в
радианах)
return b;
}
```

Файл Form1.h

```
#pragma once
#include "angle.h" //подключение заголовочного
//файла класса
double v; //глобальная переменная – значение угла

namespace cv {

    using namespace System;
    . . . . .

#pragma endregion

//Для обработчиков событий приведено только тело
//обработчика, так как заголовок формируется
//автоматически при создании обработчика

//обработчик события «Нажатие на кнопку Угол в
//градусах»
{
angle m(v);
m.print(v);
}

//обработчик события «Нажатие на кнопку Угол в
//радианах»
```

```
//результат выводится в окно сообщения
{
angle m(v);
MessageBox::Show("Значение угла в градусах=
"+m.get_rad(v),"Info",
MessageBoxButtons::OK,MessageBoxIcon::Information);
}
```

```
//обработчик события «Изменение значения в
//текстовом поле textBox1»
//чтобы создать этот обработчик можно выполнить
//двойной щелчок левой кнопкой мыши по компоненту
//textBox1 или в окне Свойства в поле события
//TextChanged
{
//преобразование содержимого textBox1 в
//вещественное число с одинарной точностью
v=Convert::ToSingle(textBox1->Text);
}
```

```
//обработчик события «Нажатие на кнопку Синус
//угла»
//результат выводится в окно сообщения
{
angle m(v);
MessageBox::Show("Значение угла в градусах=
"+m.get_sinus(v),"Info",MessageBoxButtons::OK,
MessageBoxIcon::Information);
}
```

```
//обработчик события «Нажатие на кнопку Завершение
работы»
{
Close();
}
```

Глава 4.

Наследование классов

Возможность создания новых классов на основе уже имеющихся позволяет использовать уже определенные данные и функции, при необходимости изменяя их и дополняя новыми характеристиками. Такой механизм называется наследованием и значительно упрощает процесс программирования.

Например, рассмотрим такой предмет как автомобиль. Автомобили бывают следующих типов: легковой, грузовой, автобус, троллейбус, электромобиль и т.д. Легковые в свою очередь можно разделить на седан, лимузин, хэтчбэк и другие. Они могут иметь различный тип двигателя: бензиновый, дизельный, газовый. Также автомобили можно разделить по назначению. Однако все это многообразие базируется на классе Автомобиль, так как все автомобили обладают рядом сходных черт, которые и позволяют отнести их к автомобилям. На базе класса Автомобиль мы можем получить производные классы, выбирая для этого нужные характеристики и функции, изменяя и дополняя их в зависимости от конкретной решаемой задачи.

По мере удаления от базового класса Автомобиль производные классы делаются все более специализированными. При этом они по умолчанию получают все данные и методы базового класса, то есть наследуют их. Однако производный класс должен иметь личный конструктор и деструктор, и может стать базовым для другого создаваемого класса.

Все классы C++/CLI, даже классы, которые определяет пользователь, являются производными по умолчанию. Базовым для ссылочных классов, которые мы уже рассматривали, является класс **System::Object**, то есть мы можем использовать и при необходимости переопределять его функции, например, функцию **ToString()**.

Форма записи при наследовании классов [4]:

```
ref class имя_произв_кл : сп_доступа имя_баз_класса
{ //тело производного класса };
```

Здесь **сн_доступа** указывать необязательно, так как мы будем работать со ссылочными классами, а они наследуются как открытые по умолчанию. Однако можно указать **public** явно.

При определении производного класса следует помнить, что закрытые данные базового класса доступны **только в нем и только для его функций**. Если попытаться использовать их в функциях производного класса, то возникнет ошибка. Чтобы избежать подобных ситуаций в базовом классе при объявлении данных используют спецификатор **protected**, который позволяет объявить переменные, как защищенные. Тогда они будут доступны для использования и в базовом, и в производном классе, но нигде больше. В производном классе они также будут являться защищенными.

Рассмотрим пример. Определим класс **A**, который будет использован как базовый для класса **B**.

В классе **A**:

- защищенные данные – целое **na**, вещественное **ma**,
- открытые функции:
 - конструктор по умолчанию;
 - конструктор со списком инициализации данных переменных;
 - функция **nsa()** – для сложения данных **na** и **ma**.

В классе **B**:

- закрытая переменная **kb** целого типа;
- открытые функции:
 - конструктор по умолчанию;
 - конструктор производного класса со списком инициализации с передачей аргументов из производного в базовый класс;
 - функция **nvb()** – для вычитания данных **na** и **kb**.

Создадим классы средствами среды **Microsoft Visual Studio 2008** так, как это было описано ранее. Можно поместить оба класса в один файл, а можно использовать разные файлы, создав последо-

вательно два класса и указав, при создании второго, первый в качестве базового и открытого для наследования.

Конструктор производного класса со списком инициализации и передачей аргументов из производного в базовый класс имеет определенную форму записи [4]:

КонстрПрКл (СпАрг) : БазКл (СпАрг)
{ // тело конструктора производного класса }

где ***КонстрПрКл*** – имя конструктора производного класса;
СпАрг – список аргументов;
БазКл – имя базового класса;

Конструктор производного класса **В** для рассматриваемого примера выглядит так:

```
B(int xa, float ya, int zb) : A(xa, ya) { kb=zb; }
```

Здесь в базовый класс **А** передаются первые два аргумента производного, а его третий аргумент определяется в теле конструктора производного класса.

В производном классе можно использовать те же аргументы, что и в базовом, передавать их в базовый класс напрямую, игнорируя для производного.

Ниже представлены базовый и наследующий его производный классы (наследование со словом **public**), размещенные в файле **A.h**:

```
#pragma once
ref class A
{protected:
int na;
float ma;
public:
    A(void) {}
    A(int xa, float ya) : na(xa), ma(ya) {} }
```

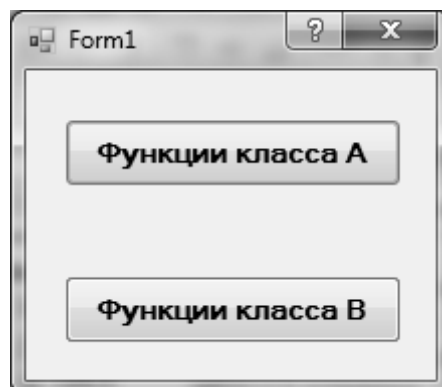
```

float nsa() {return na+ma;}
};

ref class B: public A
{
int kb;
public:
    B(void) {}
    B(int xa,float ya, int zb):A(xa,ya) {kb=zb;}
    float nvb() {return na-kb;}
};

```

Продemonстрируем использование данных классов. Создадим форму с двумя кнопками в соответствии с рисунком. Для вывода результатов в обработчике событий используем функцию **MessageBox::Show()**.



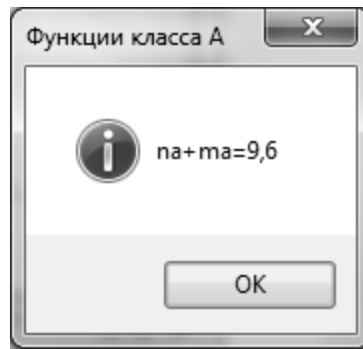
Обработчик события «Нажатие на кнопку **Функции класса А**»

```

A ob1(5,4.6); //объект ob1 базового класса A
MessageBox::Show("na+ma="+Convert::ToString(ob1.nsa())
, "Функции класса А", MessageBoxButtons::OK,
MessageBoxIcon::Information);

```

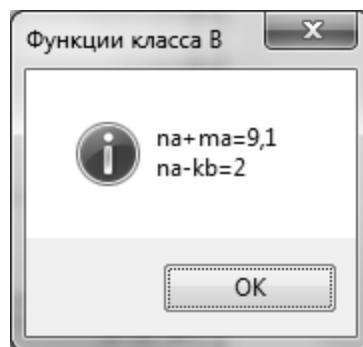
Результат выполнения обработчика события



Обработчик события «Нажатие на кнопку **Функции класса В**»

```
В ob2(7,2.1,5); //объект ob2 производного класса В
MessageBox::Show("n+m="+Convert::ToString(ob2.nsa()) +
    "\n"+"n-k="+Convert::ToString(ob2.nvb()),
    "Функции класса В", MessageBoxButtons::OK,
    MessageBoxIcon::Information);
```

Результат выполнения обработчика события



Важно помнить о необходимости использования директивы препроцессора

```
#include "A.h"
```

в файле **Form1.h**, где планируется использовать объекты данного класса.

Так как класс **A** наследуется как открытый, то его открытая функция **ns()**, будет открытой функцией производного класса **B**. Значит мы можем вызвать ее и для объекта **ob1** класса **A**, и для объекта **ob2** класса **B** в обработчиках любых событий. При этом складывать данная функция будет значения данных объекта того класса,

для которого вызвана. Результатом вызова функции **ob1.ns()** объекта, объявленного как **A ob1(5, 4.6)**, будет **9,6**. А для объекта класса **B ob2(7, 2.1, 5)** в результате вызова той же функции **ob2.ns()** получим **9,1**.

В **C++/CLI** в отличие от стандартного **C++** возможно только одиночное наследование, то есть базовым для производного класса служит один класс.

Задание для выполнения 1.

Разработать приложение, в котором в тело конструкторов и деструкторов производного и базового классов добавлен вывод сообщений об их работе. Это позволит отследить момент выполнения соответствующих конструкторов и деструкторов базового и производного классов.

Для получения результата нужно выполнить следующие шаги:

Определить базовый класс ***plane***, в котором:

данные класса целые и защищены:

v – скорость полета,

dp – дальность полета,

gr – грузоподъемность

конструктор по умолчанию,

конструктор со списком инициализации

открытые функции класса:

int vrem() – для расчета времени полета;

int get_v(), int get_dp(), int get_gr() – для вывода значений соответствующих параметров самолета;

деструктор класса.

На основе класса ***plane*** определить производный класс ***passplane***:

данные класса:

ds – пассажировместимость,

ss – ширина кресел;

конструктор по умолчанию,

конструктор со списком инициализации с передачей аргументов
в базовый класс

деструктор класса

int vrem() – переопределенная функция для расчета времени полета.

Файл plane.h

```
using namespace System::Windows::Forms;
ref class plane
{
protected:
//скорость полета, дальность
//полета, грузоподъемность
int v, dp, gr;

public:
plane():v(500), dp(5000), gr(10)
{MessageBox::Show("конструктор plane по
умолчанию");}

plane(int a1, int b1, int c1):v(a1), dp(b1), gr(c1)
{MessageBox::Show("конструктор со списком
инициализации plane");}

int vrem() {return dp/v;}

int get_v() {return v;}

int get_dp() {return dp;}

int get_gr() {return gr;}

~plane()
{MessageBox::Show("деструктор plane!");}
};
```

```

ref class passplane: public plane
{int ds, ss; // пассажировместимость, ширина кресел
public:
passplane ():plane()
{MessageBox::Show("конструктор по умолчанию
passplane!");}
passplane (int a2,int b2,int a1, int b1, int c1);
int vrem() {return (dp/v+100);}
~ passplane () {MessageBox::Show("деструктор
passplane!");} };

```

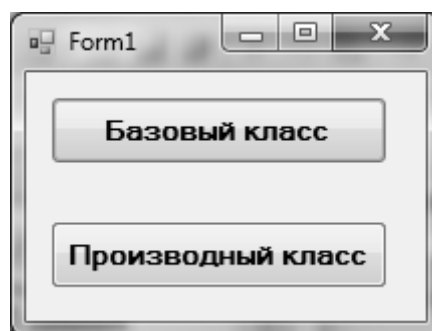
Конструктор производного класса определим в файле *plane.cpp*.

```

#include "plane.h"
passplane:: passplane (int a2,int b2,int a1,int
b1,int c1):plane(a1,b1,c1)
{
    ds=a2;
    ss=b2;
    MessageBox::Show("конструктор со списком
инициализации passplane!");}

```

На форму поместим две кнопки в соответствии с рисунком.



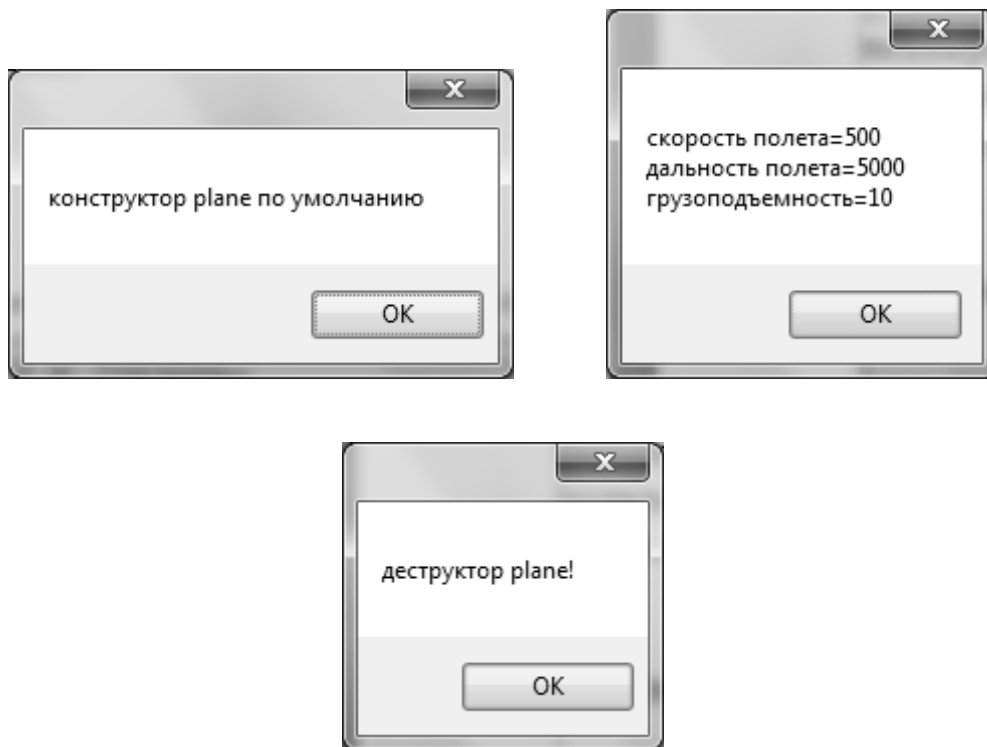
Обработчик события «Нажатие на кнопку **Базовый класс**»:

```

{ plane b;
MessageBox::Show("скорость
полета="+b.get_v()+"\ndальность полета="+
b.get_dp()+"\nгрузоподъемность="+b.get_gr()); }

```

Результат нажатия на кнопку **Базовый класс**:



Из результата видно, что при нажатии на кнопку сначала вызывается конструктор по умолчанию, в результате чего создается объект **b** класса **plane**. Затем выполняется вызов функций класса **b.get_v()**, **b.get_dp()**, **b.get_gr()** в функции **MessageBox::Show()**. И в заключении вызывается деструктор класса.

Обработчик события «Нажатие на кнопку **Производный класс**»:

```
{
passplane m(4,5,6,7,8);
MessageBox::Show("скорость
полета="+m.get_v()+"\ndальность полета="
+m.get_dp()+"\nгрузоподъемность="+m.get_gr()
+"\nбудет лететь"+m.vrem()+"ч","параметры
самолета");
}
```

Результат нажатия на кнопку **Производный класс**:



Результат представлен несколькими окнами, каждое из которых открывается после нажатия на кнопку **ОК** предыдущего. Проанализировав последовательность их появления, можно сделать вывод, что при создании объекта производного класса **passplane m(4,5,6,7,8)** первым выполняется конструктор базового класса, затем конструктор производного класса. После обращения к функциям класса в функции **MessageBox::Show()** выполняется деструктор производного класса, а затем деструктор базового класса.

Данный порядок выполнения конструкторов и деструкторов объясняется тем, что данные базового класса могут использоваться в производном классе и получают значения первыми независимо от данных производного класса. То есть конструктор базового класса вызывается первым, а конструктор производного класса – вторым.

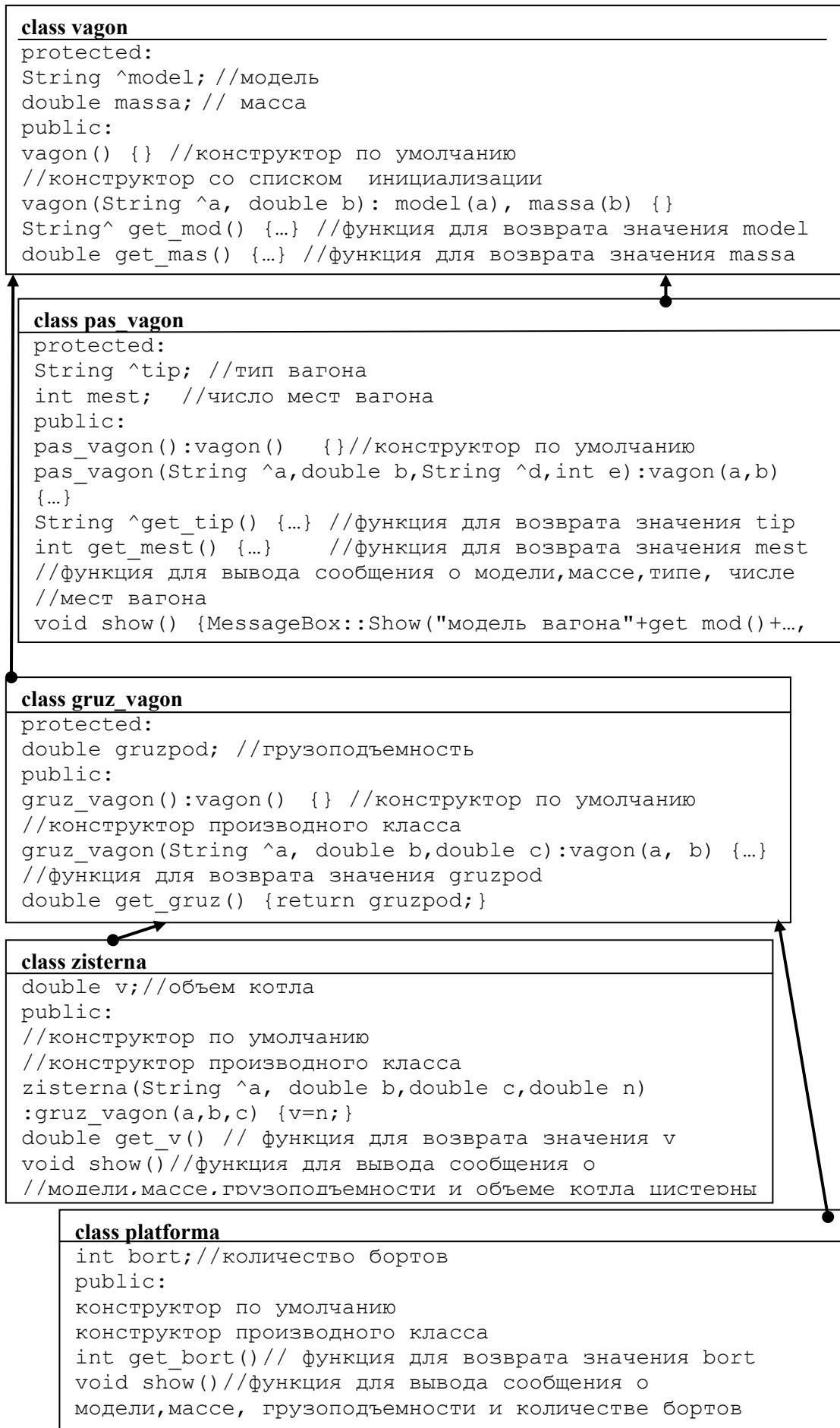
Что касается деструкторов, то последовательность их вызова обратна, так как базовый класс является основой производного, его разрушение приведет к разрушению производного.

Задание для выполнения 2.

Разработать приложение с использованием механизма наследования классов. Рекомендуемый интерфейс приложения представлен на рисунке.

Всего в приложении должно быть четыре производных класса. Все классы и функции классов определить в одном файле **vagon.h**.

Схема наследования классов представлена на рисунке. Обратите внимание, что направление стрелок – от производного класса к базовому. Вместо многоточия необходимо дописать необходимый текст программы. Для проверки правильности написания ниже приведен код для проверки. Номера компонентов **textBox** в вашем приложении могут отличаться от приведенных ниже.



Обработчик события «Нажатие на кнопку **Пассажирский вагон**»:

```
String^ ap=textBox5->Text;
double bp=Convert::ToSingle(textBox6->Text);
String^ cp=textBox7->Text;
    int dp=Convert::ToInt32(textBox8->Text);
    pas_vagon sp1(ap,bp,cp,dp);
    sp1.show();
// k - количество перевозимых людей
double k=Convert::ToSingle(textBox13->Text);
// m - количество необходимых вагонов
double m=k/sp1.get_mest();
    MessageBox::Show("Для перевозки людей
требуется: "+Math::Round(m,2)+" пассажирских
вагонов");
```

Остальные обработчики событий написать самостоятельно по аналогии с представленным выше.

Math::Round(m, 2) – округление значения переменной **m** до двух знаков после точки.

Код программы для проверки

Файл **vagon.h**

```
#pragma once
using namespace System;
using namespace System::Windows::Forms;

ref class vagon
{protected:
    String ^model;
    double massa;
public:
    vagon() {}
    vagon(String ^a, double b): model(a), massa(b)
    {}
```

```

        String^ get_mod() {return model;}
        double get_mas() {return massa;}
};

```

```

ref class груз_vagon :public vagon
{ protected:
    double грузpod;
public:
    груз_vagon():vagon() {}
    груз_vagon(String ^a, double b,double c):
    vagon(a, b) {грузpod=c;}
    double get_gruz() {return грузpod;}
};

```

```

ref class zisterna :public груз_vagon
{double v;
public:
    zisterna():груз_vagon() {}
    zisterna(String ^a, double b,double c,double n):
        груз_vagon(a,b,c) {v=n;}
    double get_v() {return v;}
};

```

```

void show()
{MessageBox::Show(model+" "+get_mas()+"
"+get_gruz()+" "+get_v(),"ИИСТЕРФА");}
};

```

```

ref class platforma :public груз_vagon
{int bort;
public:
    platforma():груз_vagon() {}
    platforma(String ^a, double b,double c,int
n):груз_vagon(a,b,c) {bort=n;}
    int get_bort() {return bort;}
    void show()
};

```

```
{MessageBox::Show(get_mod()+" "+get_mas()+"
"+get_gruz()+" "+get_bort(),"ПЛАТФОРМА");}
};
```

```
ref class pas_vagon:public vagon
{protected:
String^ tip;
    int mest;
public:
pas_vagon():vagon() {}
pas_vagon(String ^a, double b,String ^d,int e
):vagon(a, b) {tip=d; mest=e; }
String ^get_tip() {return tip;}
int get_mest() {return mest;}
void show()
{MessageBox::Show("модель вагона "+get_mod()+"
"+"\\nмасса вагона "+get_mas()+" т"+"\\nтип вагона
"+
                        get_tip()+"\\nчисло мест
"+get_mest(),"ПАССАЖИРСКИЙ");}
};
```

Обработчик события «Нажатие на кнопку Платформа»:

```
String^ a=textBox9->Text;
double b=Convert::ToSingle(textBox10->Text);
double c=Convert::ToSingle(textBox11->Text);
    int d=Convert::ToInt32(textBox12->Text);
platforma sp2(a,b,c,d);
sp2.show();
```

Обработчик события «Нажатие на кнопку Цистерна»:

```
String^ az=textBox1->Text;
double bz=Convert::ToSingle(textBox2->Text);
double cz=Convert::ToSingle(textBox3->Text);
```

```

double dz=Convert::ToSingle(textBox4->Text);
zisterna s3(az,bz,cz,dz);
s3.show();
//k количество необходимой солярки
double k=Convert::ToInt32(textBox15->Text);
double m=k/s3.get_v(); //количество цистерн
MessageBox::Show("Для перевозки солярки требуется:
"+Math::Round(m,1)+" цистерн");

```

Задание для выполнения 3.

Разработать приложение с использованием механизма наследования классов. Рекомендуемый интерфейс приложения представлен на рисунке.

Определите базовый класс **korabli**, в состав которого включены защищенные данные:

```

double dl;//длина
double sh;//ширина

```

```
double vdi;//водоизмещение
double sk;//скорость
double mo;//мощность
int ek;//экипаж
String^ vv;//вид вооружения
```

и функции :

конструктор класса со списком инициализации

```
double sp()/*для расчета шпации 0.002*d1+0.48*/
double ki()/*для расчета ширины горизонтального
киля 800 + 5*d1 */
```

Определите два производных класса **podvodn** (подводные лодки) и **kreiser** (крейсера) .

Состав класса **podvodn**:

Данные:

```
double gp; //глубина погружения
String^ ie; //источник энергии
```

Функции:

конструктор класса с передачей аргументов из производного в базовый класс

```
void show()/*Функция для вывода сообщения о длине, ширине, водоизмещении, скорости, мощности, экипаже, виде вооружения, глубине погружения, источнике энергии, значении шпации, ширине горизонтального киля*/
```

Состав класса **kreiser**:

Данные:

```
double os;//осадка
String^ av;//авиационное вооружение
```

Функции:

Конструктор класса с передачей аргументов из производного в базовый класс

```
void show()/*Функция для вывода сообщения о длине, ширине, водоизмещении, скорости, мощности, экипаже, виде вооружения, осадке, авиационном вооружении*/
```

ния, осадке, авиационном вооружении, значении шпации, ширине горизонтального киля*/

Для данного класса, определенного в файле **korabli.h**, напишите программу, демонстрирующую ввод исходных данных и вывод параметров подводной лодки и крейсера в окне сообщения (MessageBox::Show) по нажатию соответствующих кнопок.

**Код программы для проверки
(номера компонентов в вашем приложении могут отличаться)**

Файл Form1.h

```
#pragma once
#include " korabli.h"
```

```
. . .
```

Обработчик события «Нажатие на кнопку
ХАРАКТЕРИСТИКИ ПОДВОДНОЙ ЛОДКИ»

```
double ak=Convert::ToDouble(textBox1->Text);
double bk=Convert::ToDouble(textBox2->Text);
double ck=Convert::ToDouble(textBox3->Text);
double dk=Convert::ToDouble(textBox4->Text);
double eh=Convert::ToDouble(textBox5->Text);
int fk=Convert::ToInt16(textBox6->Text);
String^ g=textBox7->Text;
double hk=Convert::ToDouble(textBox8->Text);
String^ k=textBox9->Text;
    podvodn vk1(ak,bk,ck,dk,eh,fk,g,hk,k);
        vk1.show();
```

Обработчик события «Нажатие на кнопку
ХАРАКТЕРИСТИКИ КРЕЙСЕРА»

```
double ac=Convert::ToDouble(textBox10->Text);
```



```

double bc=Convert::ToDouble(textBox11->Text);
double cc=Convert::ToDouble(textBox12->Text);
double dc=Convert::ToDouble(textBox13->Text);
double ehc=Convert::ToDouble(textBox14->Text);
int fc=Convert::ToInt16(textBox15->Text);
String^ gc=textBox16->Text;
double hc=Convert::ToDouble(textBox17->Text);
String^ kc=textBox18->Text;
    kreiser vk2(ac,bc,cc,dc,ehc,fc,gc,hc,kc);
    vk2.show();}

```

//Определение классов

Файл korabli.h

```

#pragma once
using namespace System;
using namespace System::Windows::Forms;
ref class korabli
{protected:
    double dl;//длина
    double sh;//ширина
    double vdi;//водоизмещение
    double sk;//скорость
    double mo;//мощность
    int ек;//экипаж
    String^ vv;//вид вооружения
public:
    korabli(double ak,double bk,double ck,double
dk,double ек,int fk,String^ zk):dl(ak),sh(bk),
vdi(ck),sk(dk),mo(ек),ек(fk),vv(zk) {}
    double sp()
    {return 0.002*dl+0.48;}
    double ki()
    {return 800 + 5*dl;}
};

ref class podvodn:korabli

```

```

{    double gp; //глубина погружения
    String^ ie; //источник энергии
public:
    podvodn(double ak,double bk,double ck,double
dk,double ek,int fk,String^ zk,double gk,String^
hk):korabli(ak,bk,ck,dk,ek,fk,zk) {gp=gk; ie=hk;}
    void show()
    {MessageBox::Show("длина="+dl+"\nширина"+sh+"\n
водоизмещение="+vdi+"\nскорость="+sk+"\nмощность=
"+mo+"\nэкипаж="+ek+"\nвид
вооружения="+vv+"\nглубина
погружения="+gp+"\nисточник
энергии="+ie+"\nшпация="+sp()+"\nширина
горизонтального киля="+ki(),"ПОДВОДНАЯ
ЛОДКА",MessageBoxButtons::OK);    } };

//крейсера эсминцы авианосцы подводные лодки
ref class kreiser:korabli
{double os; //осадка
String^ av; //авиационное вооружение
public:
kreiser(double ak,double bk,double ck,double
dk,double ek,int fk,String^ zk,double nk,String^
mk):korabli(ak,bk,ck,dk,ek,fk,zk) {os=nk; av=mk;}
void show()
    {MessageBox::Show("длина="+dl+"\nширина"+sh+"\n
водоизмещение="+vdi+"\nскорость="+sk+"\nмощность=
"+mo+"\nэкипаж="+ek+"\nвид вооружения="+vv+
"\nосадка=" +os+ "\navиационное
вооружение="+av+"\nшпация="+sp()+"\nширина
горизонтального
киля="+ki(),"КРЕЙСЕР",MessageBoxButtons::OK); } };

```

Глава 5.

Полиморфизм

Полиморфизм – свойство, позволяющее использовать одно имя для обозначения действий, общих для родственных классов[2]. Уточнение выполняемых действий осуществляется в зависимости от типа и количества обрабатываемых данных. Основные формы полиморфизма:

- перегрузка функций и операций;
- виртуальные функции;
- обобщенные функции и классы (шаблоны).

Перегрузку функций и операций относят к *статическому* полиморфизму, так как он поддерживается на этапе компиляции. Виртуальные функции соответствуют *динамическому* полиморфизму, поскольку он реализуется при выполнении программ.

Достоинством полиморфизма является то, что позволяет использовать многократно один раз составленные алгоритмы и, как следствие, обеспечивает уменьшение избыточного кода.

5.1. Перегрузка функций

При разработке программ на C++/CLI можно использовать нескольких функций с одинаковым именем, выполняющих похожие действия над данными *разных типов*. Например, в программе определены две функции с прототипами:

```
int fsum(int, int, int);  
float fsum(float, float);
```

В процессе компиляции программы при обращении к функциям ***fsum()*** в зависимости от типа и числа аргументов будет осуществляться выбор требуемого экземпляра функции. Таким образом реализуется перегрузка функций.

При определении в программе нескольких одноименных функций при компиляции возможны следующие варианты [4]:

1. *Возвращаемые значения, тип и число параметров нескольких функций совпадают.* В данном случае имеет место переобъявление функции во втором и последующем объявлении:

```
int fh(int ky,int xy);
int fh(int bz,int az);
```

2. *Тип и число параметров одноименных функций совпадают, но возвращаемые значения различны. При этом возникает ошибка компиляции.*

```
int fq(int,int,int);
float fq(int,int,int);
```

3. *Тип и число параметров одноименных функций различны. Такие функции считаются перегружаемыми.*

```
int fsum(int rb,int an,int nz)
{return rb+an+nz;}
float fsum(float n, float v)
{return n+v;}
```

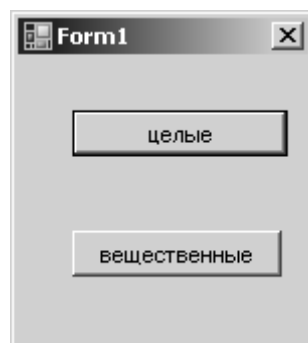
Задание для выполнения 1.

Разработать приложение с использованием перегрузки функций *rez()* при различных типах (*int* ,*float*) и количестве (2, 3) аргументов.

$$rez(int\ xb,int\ mb)=\begin{cases}xb-mb+100, & \text{если } xb \cdot 2 > mb \\ 100 \cdot xb - mb, & \text{если } xb \cdot 2 \leq mb\end{cases}$$

$$rez(float\ xa, float\ ma, float\ ka)=\begin{cases}xa + ma \cdot ka, & \text{если } xa > (ka + 100 - ma \cdot xa) \\ xa - ma / ka, & \text{если } xa \leq (ka + 100 - ma \cdot xa)\end{cases}$$

Рекомендуемый интерфейс приложения:



Определить функции в файле **Form1.h** после строки

```
#pragma once  
float rez(float xa, float ma, float ka)  
{ if (xa>(ka+100-ma*xa)) return xa+ma*ka;  
else return xa-ma/ka;}  
  
int rez(int xb, int mb)  
{if (xb*2>mb)return xb-mb+100;  
else return 100*xb-mb;}
```

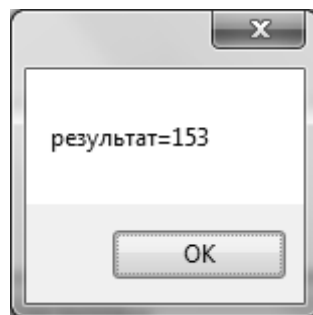
В тексте обработчиков событий объявить переменные, которые будут использоваться в качестве аргументов функции, вызов функции осуществить при выводе окна сообщения

```
MessageBox::Show("результат="+. . .);
```

Обработчик события «Нажатие на кнопку **Целые**»

```
{ int az=65,bz=12;  
  MessageBox::Show("результат="+rez(az,bz)); }
```

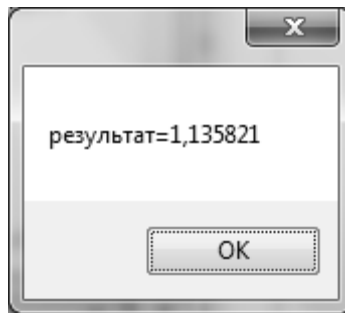
Результат:



Обработчик события «Нажатие на кнопку **Вещественные**»

```
{ float wa=2.3,ac=7.8,uz=6.7;  
  MessageBox::Show("результат="+rez(wa,ac,uz)); }
```

Результат:



Примером перегрузки функций является перегрузка конструктора класса.

5.2. Перегрузка операций

При *перегрузке стандартных операций* на этапе компиляции вариант реализации операции выбирается в зависимости от типов данных, в ней участвующих. Так операцию сложения можно переопределить для объединения строк, сложения комплексных чисел, данных, представленных в виде метров и сантиметров, часов и минут и так далее.

Условием перегрузки операций является участие в операции хотя бы одного объекта пользовательского класса.

Формат переопределения операции [4]:

тип_результата operator знак_операции (список параметров)
{операторы ;}

В качестве типа результата подразумевается тип значения, являющегося результатом данной функции. **operator** – это ключевое слово. Знак операции может быть +, −, /, *, ++ и т.д. В списке параметров указываются те, при которых выбирается данный вариант перегрузки операции.

Если функция перегрузки оператора является функцией класса, то при ее определении вне класса нужно указать перед ключевым словом **operator** принадлежность к классу: **имя_класса :: operator**.

Рассмотрим классический пример использования перегрузки операций [2]. Определим класс, представляющий длину в футах и дюймах, и определим в нем перегрузку операции сложения и вычитания.

```

using namespace System;
ref class dlina
{int ft; // футы
int dm; // дюймы
public:
    dlina(int x,int y):ft(x),dm(y) {}//конструктор
//переопределение функции ToString(), чтобы
//выводить объекты класса dlina текстовой строкой

virtual String^ ToString() override
    {return ft+"футов"+dm+"дюймов";}

// Реализация функции operator+
// перегрузка операции сложения для двух объектов
// класса dlina

    dlina^ operator+(dlina^ dl)
    {int dmt=dm+dl->dm+12*(ft+dl->ft);
    return gcnew dlina(dmt/12,dmt%12);    }

    dlina^ operator-(dlina^ dl)
    {int dmt=dm-dl->dm+12*(ft-dl->ft);
    return gcnew dlina(dmt/12,dmt%12);    }
};

```

Результатом работы функции **operator+** будет объект класса **dlina**, характеризующийся данными в виде длины, представленной в футах (**ft**) и дюймах (**dm**). В теле функции сначала происходит сложение длин двух объектов. Причем дюймы складываются с дюймами и прибавляются к сумме футов, переведенной в дюймы путем умножения на **12** (количество дюймов в футах).

При создании нового объекта, как результата сложения исходных, футы определяются в виде результата целочисленного деления общей суммы на **12**, а дюймы — как остаток от деления данных чисел.

Обработчик событий:

```
{
    dlina^ d1=gcnew dlina(2,6);
    dlina^ d2=gcnew dlina(3,8);
    dlina^ t=d1+d2;
    MessageBox::Show("общая длина =" +t);
}
//вызывается переопределенная функция ToString()
//класса String

dlina^ d1=gcnew dlina(12,3);
dlina^ d2=gcnew dlina(3,5);
dlina^ t=d1-d2;
MessageBox::Show("разница в длине =" +t); }
```

Использованная в тексте обработчика событий

```
dlina^ d1=gcnew dlina(12,3);
```

gcnew создает экземпляр класса в памяти с помощью конструктора и передает ссылку на него в переменную **^d1**. Только после этого можно использовать функции класса, включая функции, выполняющие перегрузку операций, для объектов данного класса.

В результате выполнения обработчика события последовательно выведется два окна со значениями суммы и разницы объектов класса **dlina**.

Задание для выполнения 2.

Разработать приложение с использованием перегрузки операций, обеспечивающее возможность сложения и вычитания данных о времени пробега, представленном в часах, минутах, секундах.

Определить класс **estafeta**, в состав которого входят данные:

```
int hr; //время в часах
```



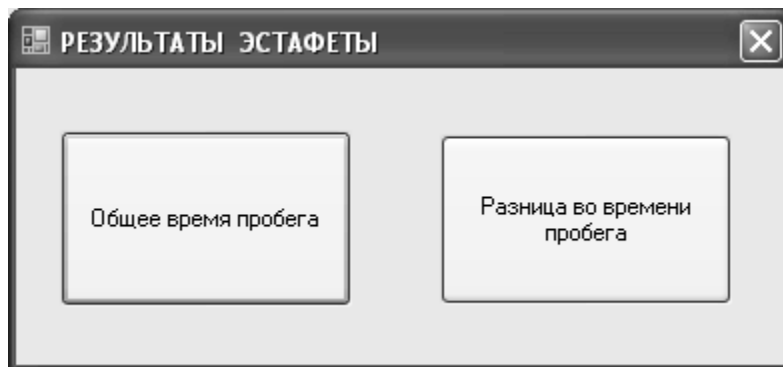
```
int mn; //время в минутах
int sc; //время в секундах
функции:
```

- конструктор со списком инициализации
- конструктор без параметров
- переопределение функции **ToString()** класса **String**, чтобы выводить объекты класса **estafeta** текстовой строкой:

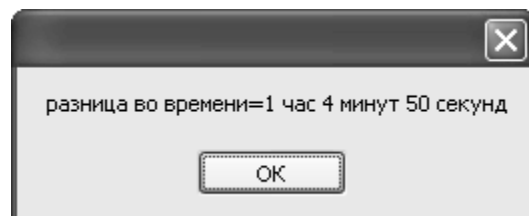
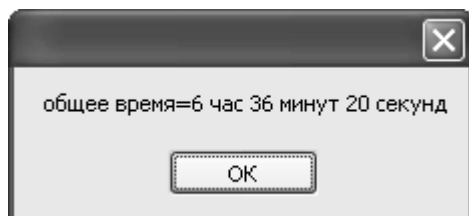
```
virtual String^ ToString() override {...}
```

- перегрузка операции сложения (+) для объектов класса **estafeta**
- перегрузка операции вычитания (-) для объектов класса **estafeta**.

Рекомендуемый интерфейс приложения представлен на рисунке.



Результат нажатия на кнопки



Для вывода сообщения в обработчиках событий используйте функцию

```
MessageBox::Show(" . . . ");
```

Код программы для проверки

Файл **estafeta.h**

```
#pragma once
using namespace System;
using namespace System::Windows::Forms;

ref class estafeta
{
    int hr;
    int mn;
    int sc;
public:
    estafeta() {}
    estafeta(int x, int y, int z):hr(x),mn(y),sc(z)
    {}

    virtual String^ ToString() override
    {
        return hr+" час "+mn+" минут "+sc+" секунд ";
    }

    estafeta^ operator+ (estafeta^ vr)
    {
        int vrt=(hr+vr->hr)*3600+(mn+vr->mn)*60+sc+vr->sc;
        return gcnew
            estafeta(vrt/3600,(vrt%3600)/60,((vrt%3600)%60));
    }

    estafeta^ operator- (estafeta^ vr)
    {
        int vrt=(hr-vr->hr)*3600+(mn-vr->mn)*60+sc-vr->sc;
        return gcnew
            estafeta(vrt/3600,(vrt%3600)/60,((vrt%3600)%60));
    }
};
```

Не забудьте использовать в файле **Form1.h** директиву препроцессора:

```
#include "estafeta.h"
```

Обработчик события «Нажатие на кнопку **Общее время пробега**»

```
estafeta^ n=gcnew estafeta(3, 50, 35);  
estafeta^ g=gcnew estafeta(2, 45, 45);  
estafeta^ m=n+g;  
MessageBox::Show("общее время="+m);
```

Обработчик события «Нажатие на кнопку **Разница во времени пробега**»

```
estafeta^ n=gcnew estafeta(3, 50, 35);  
estafeta^ g=gcnew estafeta(2, 45, 45);  
estafeta^ m=n-g;  
MessageBox::Show("разница во времени="+m);
```

5.3. Виртуальные функции

Виртуальная функция представляет собой функцию базового класса, которая переопределяется в производном классе.

При объявлении виртуальной функции в базовом классе перед ее именем указывается ключевое слово ***virtual***, которое не обязательно при переопределении данной функции в производном классе. Переопределяемая виртуальная функция должна иметь те же тип, число параметров и тип возвращаемого значения, но может выполнять действия, отличные от одноименной функции базового класса.

Виртуальную функцию можно вызывать как обычную функцию класса. Однако использование отслеживаемого дескриптора и вызов виртуальной функции с его помощью обеспечивает реализацию динамического полиморфизма, то есть выбор используемой виртуальной функции выполняется при выполнении программы.

Если на базе класса, имеющего виртуальную функцию, создано несколько производных классов, то для каждого из объектов данных классов вызывается свой вариант виртуальной функции.

Данный факт подтверждается при рассмотрении приведенного ниже примера, где функция **show()** определена как виртуальная в ба-

зовом и переопределена в двух производных классах.

```
using namespace System;
using namespace System::Windows::Forms;
ref class virt
{protected:
double dl;//длина
double sh;//ширина
public:
virt(double a,double b):dl(a),sh(b) {}
double sp()
{return 0.002*dl+0.48;}
virtual void show(){MessageBox::Show("Базовый
класс");}
};

ref class podvodn:virt
{
    double gp; //глубина погружения
    public:
    podvodn(double a,double b,double c):virt(a,b)
    {gp=c;}

    virtual void show()override
    {MessageBox::Show("длина="+dl+"\nширина"+sh+"\nспа
ция="+sp()+"\nглубина погружения="+gp,"ПОДВОДНАЯ
ЛОДКА",MessageBoxButtons::OK);    }
};

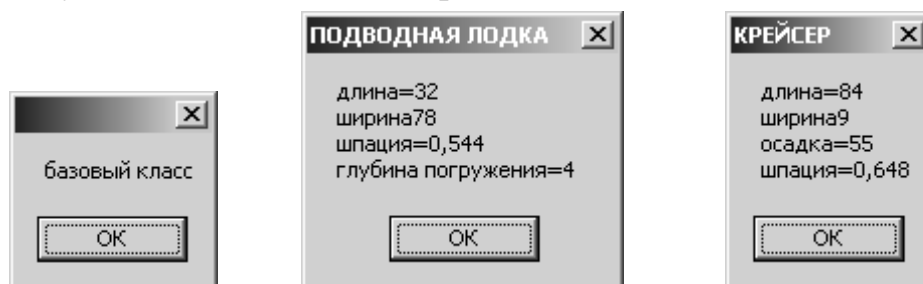
ref class kreiser:virt
{
double os;//осадка
public:
kreiser(double a,double b,double c):virt(a,b)
{os=c;}
virtual void show()override
```

```
{MessageBox::Show("длина="+dl+"\nширина"+sh+"\n\nосадка="+os+"\n\nшпация="+sp(),"КРЕЙСЕР",MessageBoxButtons::OK);}
```

Обработчик события:

```
{ virt^ d1=gcnew virt(12,3);
podvodn^ d2=gcnew podvodn(32,78,4);
kreiser^ d3=gcnew kreiser(84,9,55);
d1->show();
d2->show();
d3->show();}
```

Результаты выполнения обработчика события:



Полученные результаты показывают, что при обращении к виртуальной функции производного класса происходит вызов функции производного класса. При использовании обычных функций, то вызывалась бы одноименная функция базового класса.

В ссылочном классе, используемом в качестве базового, могут присутствовать виртуальные функции, не выполняющие никаких действий, и подлежащие обязательному переопределению в производном классе.

Подобные виртуальные функции, неопределенные в ссылочном классе называются абстрактными и объявляются с применением ключевого слова `abstract` в виде прототипов:

```
virtual void show()abstract;
```

Класс, который содержит одну или более абстрактных функций, должен быть явно объявлен как абстрактный с помощью ключевого слова `abstract`, следующего за именем класса. Так как такой

класс имеет функцию, у которой отсутствует тело функции, то ни одного объекта этого класса создать нельзя. Как правило, он используется как базовый для наследования производными классами.

```
ref class virt abstract
{
    -||-
virtual void show() abstract;
};
```

Задание для выполнения 3.

Разработать приложение с использованием виртуальных функций. Создать базовый класс **pp**:

данные:

две размерности фигуры: **double st1, st2**(защищенные).

функции:

конструктор класса

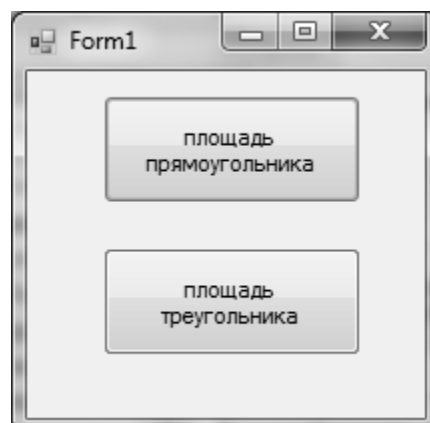
две виртуальные функции:

- **rez()**, которая в базовом классе **pp** возвращает площадь прямоугольника, а при ее переопределении в производном классе **pt** возвращает площадь прямоугольного треугольника,

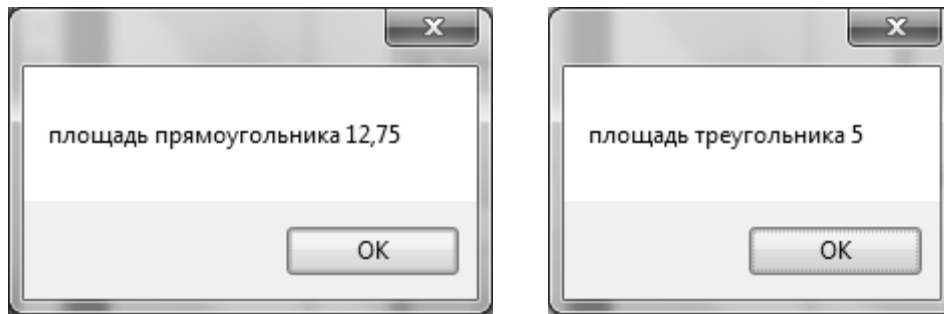
- функция **show()** для вывода рассчитанного значения для базового и производного класса.

Базовый и производный класс определите самостоятельно (важно использовать **override** при переопределении виртуальных функций в производном классе).

Рекомендуемый интерфейс приложения приведен на рисунке.



Результат нажатия на соответствующие кнопки:



Код программы для проверки

Файл pp.h

```
#pragma once
using namespace System::Windows::Forms;
ref class pp
{protected:
double st1,st2;
public:
    pp(double a, double b):st1(a),st2(b) {}
    virtual double rez() {return st1*st2;}
    virtual void show(){MessageBox::Show("площадь
прямоугольника "+rez());}
};

ref class pt:pp
{public:
    pt(double a, double b):pp(a,b) {}
    virtual double rez() override
    {return 0.5*st1*st2;}
    virtual void show() override
    {MessageBox::Show("площадь треугольника
"+rez());}
};
```

Обработчик события «нажатие на кнопку **площадь
прямоугольника**»:

```
{pp s1(2.5,5.1);
s1.show();}
```

Обработчик события «нажатие на кнопку **площадь треугольника**»:

```
{pt s2(2,5);
s2.show();}
```

Задание для выполнения 4.

Разработать приложение с использованием виртуальных функций.

Определить абстрактный базовый класс `transport`, данными которого являются:

защищенные:

```
int m; //масса
```

```
double v, s; //скорость, расстояние
```

```
String^ model; //модель
```

```
double obak; //объем бака
```

```
double topl; //расход топлива
```

открытые:

конструктор со списком инициализации

конструктор без параметров

```
virtual double t() abstract; //функция расчета
времени передвижения
```

```
virtual double z() abstract; //функция расчета запаса
хода по топливу
```

```
virtual void show() abstract; //функция для вывода
сообщения о параметрах объекта
```

Определить производный от класса `transport` класс `bus`, данными которого являются:

```
{int klb; //количество посадочных мест
```

```
int to; //количество дверей
```



```
int pr;//объем багажных отсеков
```

открытые:

конструктор класса с передачей аргументов из производного в базовый класс

переопределенные виртуальные функции:

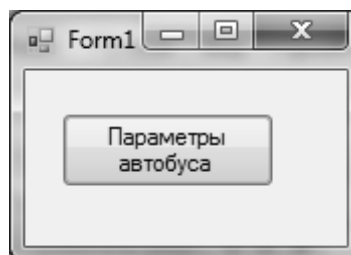
```
virtual double t() override  
{...}
```

```
virtual double z() override  
{return obak/topl;}
```

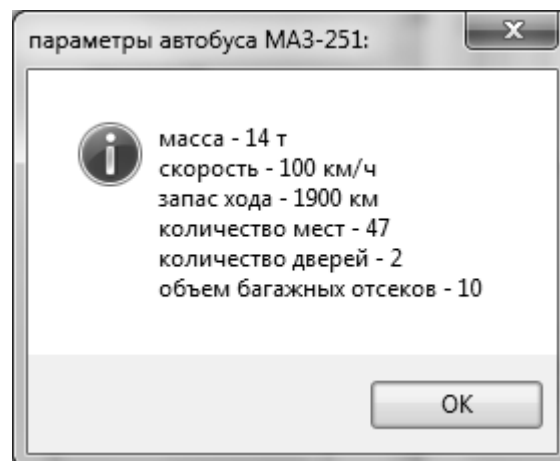
```
virtual void show()override {. . .}
```

Для данного класса, определенного в файле **transport.h**, определите обработчик события, где создается объект класса и для него вызывается функция **show()**.

Рекомендуемый интерфейс приложения представлен на рисунке.



Результат нажатия на кнопку



Код программы для проверки

Обработчик события «нажатие на кнопку **Параметры автобуса**»:

```
{bus n(14,100,600,"МАЗ-251",500,26,47,2,10);  
n.show();}
```

Файл transport.h

```
#pragma once  
using namespace System;  
using namespace System::Windows::Forms;  
ref class transport abstract  
{protected:  
    int m;//масса  
    double v, s; //скорость, расстояние  
    String^ model; //модель  
    double obak; //объем бака  
    double topl; //расход топлива  
  
public:  
    transport(void) {}  
    transport(double a, double b, double c,  
String^ mod, double ob,double tp):m(a),v(b),s(c),  
model(mod),obak(ob),topl(tp) {}  
  
virtual double t() abstract;//функция расчета  
времени передвижения  
virtual double z() abstract;//функция расчета  
запаса хода по топливу  
virtual void show() abstract;//функция для вывода  
сообщения о параметрах объекта  
};  
  
ref class bus:transport  
{int klb;//количество посадочных мест  
int to;//количество дверей
```

```

int pr;//объем багажных отсеков
public:
    bus(int a, double b, double c, String^
mod,double ob, double tp, int k, int t, int
p):transport(a,b,c,mod,ob,tp)

    {klb=k;
      to=t;
      pr=p;}
virtual double t() override
{return s/v;}

virtual double z() override
{return Math::Round((obak/topl))*100;}

virtual void show()override
{
    MessageBox::Show("масса - "+m+" т+"\nскорость
- "+v+" км/ч+"\nзапас хода - " + z()+"
км+"\nколичество мест - "+klb+" \nколичество
дверей - "+to+"\nобъем багажных отсеков -
"+pr,"параметры автобуса "+model+":",
MessageBoxButtons::OK,MessageBoxIcon::Information);
}};

```

5.4. Шаблоны

Выделяют шаблоны функций и шаблоны классов. Они позволяют использовать функции и классы для различных типов данных.

Шаблон функции имеет следующий формат записи [5]:

```

template <class T>
T имя_функции(список_параметров)
{
// тело функции
}

```

template – это ключевое слово, обозначающее, что создается шаблон. **class** – ключевое слово для объявления имени типа. *T* – заместитель имени типа, вместо которого при вызове функции будет использоваться реальный тип данных, применяемый при компиляции. В теле функции записывается код, определяющий действия, этой функцией выполняемые.

Задание для выполнения 5.

Создать и использовать шаблон функции **fs1(ns,ms,hs,as)** для вычисления суммы $y = \sum_{x=ns}^{ms} (x \cdot as + b)$, где *x* изменяется от **ns** до **ms** с шагом **hs**, переменная **b** определяется по формуле:

$$b = \begin{cases} as - x, & \text{если } x \geq 20 \\ x + as, & \text{если } x < 20 \end{cases}.$$

Рекомендуемый интерфейс приложения представлен на рисунке:



Для выполнения задания необходимо выполнить следующие действия:

- Определить шаблон функции в файле **func. h**, созданном с помощью команды главного меню **Проект/Добавить новый элемент...** В открывшемся диалоговом окне необходимо выбрать **Заголовочный файл (.h)** и ввести имя **func**.
- В открывшемся окне редактора кода файла **func. h** определить шаблон функции **fs1(ns,ms,hs,as)**:

```

template <class T>
T fs1( T nt, T mt, T ht, T at)
{
T b,x=nt,zt=0;
while(x<=mt)
{if (x>=20) b=at;
else b=x+at;
zt=zt+x*at+b;
x=x+ht;}
return zt;
}

```

– Подключить созданный файл к **Form1.h** после

```
#pragma once
```

```
#include " func.h"
```

– Использовать созданный шаблон для обработки целых и вещественных чисел:

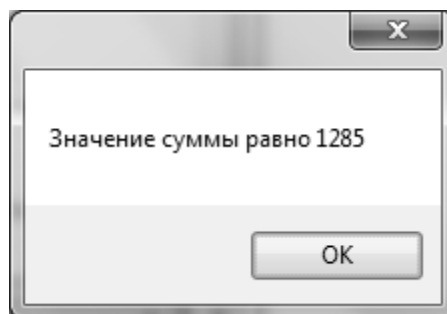
Обработчик события «Нажатие на кнопку *int*»:

```

int nz=10,mz=100,hz=20,az=5;
MessageBox::Show("Значение суммы равно " +
fs1(nz,mz,hz,az));

```

Результат выполнения события



Обработчик события «Нажатие на кнопку *float*»:

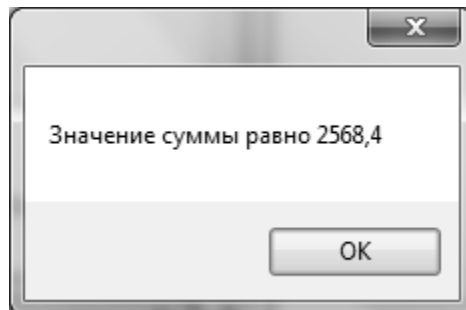
```
float nv=5.1,mv=99.6,hv=10.2,av=4.9;
```

```

        MessageBox::Show ("Значение суммы равно " +
fs1 (nv,mv,hv,av) );

```

Результат выполнения события



Шаблоны класса

Шаблон класса, как и шаблон функции, предназначен для обработки данных различных типов и имеет похожий формат записи [5]:

```

template <class T>
ref class имя_класса
{
    // тело класса;
}

```

Рассмотрим простой пример шаблона класса, определенного в заголовочном файле, по аналогии с примером, использующим шаблоны функции.

```

template <class T>
ref class MyT
{
    T tx,ty;
public:
    MyT(T at, T bt):tx(at),ty(bt) {}
    T del() {return tx/ty;}
};

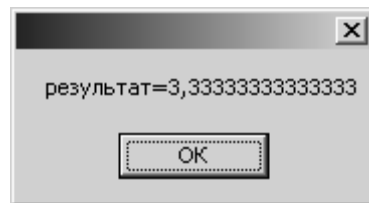
```

Обработчик события «Нажатие на кнопку»

//Создаем версию класса MyT для вещественных значений

```
MyT <double> dob(10.0, 3.0);  
MessageBox::Show("результат="+dob.del ());
```

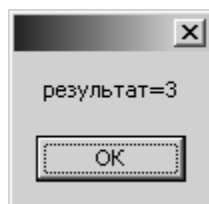
Результат выполнения обработчика событий:



Обработчик события «Нажатие на кнопку»

```
//Создаем версию класса MyT для целых значений  
MyT <int> dob1(10, 3);  
MessageBox::Show("результат="+dob1.del ());
```

Результат выполнения обработчика событий:



Результаты подтверждают использование деления вещественных чисел при использовании объекта типа **double** и целочисленного деления для объекта типа **int**.

Обратите внимание на особенность объявления объектов класса с использованием шаблона. Используемый в конкретном случае тип данных указывается в угловых скобках перед именем объекта создаваемого класса:

```
MyT <double> dob(10.0, 3.0);
```

```
MyT <int> dob1(10, 3);
```

Задание для выполнения 6.

Разработать приложение, используя приведенный выше пример, с использованием шаблона класса (синоним – параметризованного, родового, обобщенного класса), в котором имеются:

данные: *tx*, *ty*, *tz*;

конструктор класса;

функции:

– *rezt()*, в результате работы которой рассчитывается сумма **15** элементов ряда, где каждый элемент ряда определяется по формуле:

di=tx + ty/tz;

– *void show()* – рассчитанное значение выводится с помощью окна сообщения *MessageBox::Show("... ")*.

Рекомендуемый интерфейс приложения



Для выполнения задания необходимо выполнить следующие шаги:

– Определить шаблон функции в файле **ks. h**, созданном с помощью команды главного меню **Проект/Добавить новый элемент...**, выбрать **Заголовочный файл (.h)**.

```
template <class T> ref class Sm
{
T tx,ty,tz;
public:
```

//конструктор со списком инициализации

//функция для расчета суммы ряда

```
T rezt()
{
T di=0;
. . . }
```


//вывод сообщения: сумма элементов ряда равна ...

```
void show()  
    {MessageBox::Show(. . .)}
```

– Подключить созданный файл к **Form1.h** после

```
#pragma once  
#include "ks.h"
```

– В тексте обработчиков событий создать объекты классов, использовать данные типов *int* и *double*, функцию *show()*.

Код программы для проверки

Шаблон класса

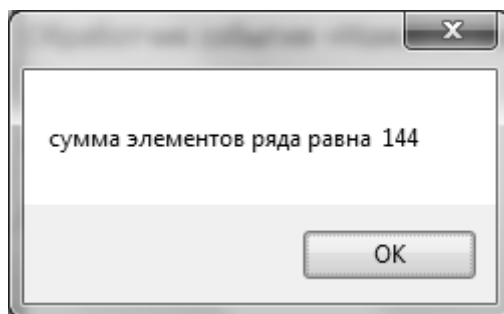
```
using namespace System;  
using namespace System::Windows::Forms;  
  
template <class T> ref class Sm  
{  
    T tx,ty,tz;  
public:  
    //конструктор со списком инициализации  
    Sm(T at, T bt, T ct):tx(at),ty(bt),tz(ct) {}  
  
    //функция для расчета суммы ряда  
    T rezt()  
    {T di=0;  
    for(int it=0;it<=15;it++)  
        di=di+tx+ty/tz;  
    return di;}  
  
// вывод сообщения: сумма элементов ряда равна  
void show()  
{  
    MessageBox::Show("сумма элементов ряда
```

```
равна "+Math::Round(rezt(), 2));  
    }  
};
```

Обработчик события «Нажатие на кнопку **int**»

```
Sm <int> oba(9, 3, 4);  
oba.show();
```

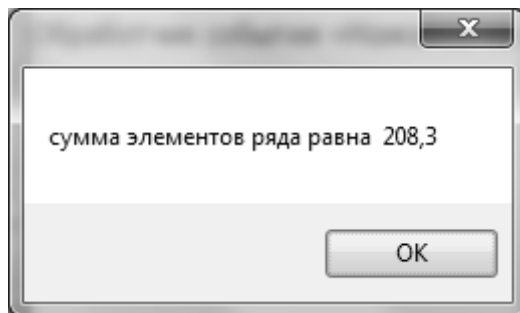
Результат выполнения события



Обработчик события «Нажатие на кнопку **double**»

```
Sm <double> obb(12.5, 2.8, 5.4);  
obb.show();
```

Результат выполнения события



Глава 6.

Использование расширенных элементов управления

6.1. Компоненты список (ListBox, ComboBox), флажок (CheckBox) и переключатель (RadioButton)

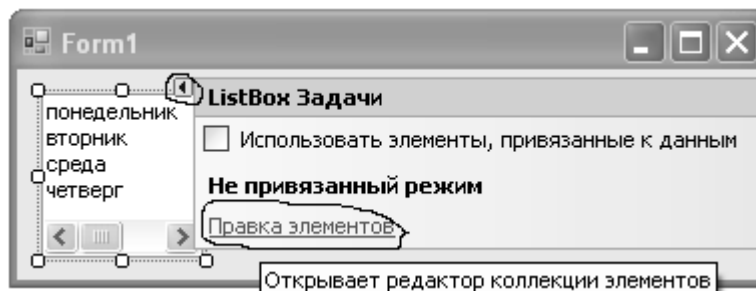
Компоненты **ListBox**, **ComboBox**, **CheckBox** и **RadioButton** находятся в окне **Панель элементов** в группе **Стандартные элементы управления**. Многие свойства, события и методы данных компонентов характерны для рассмотренных ранее. Однако имеются и свойства, присущие только им.

Компонент **ListBox**

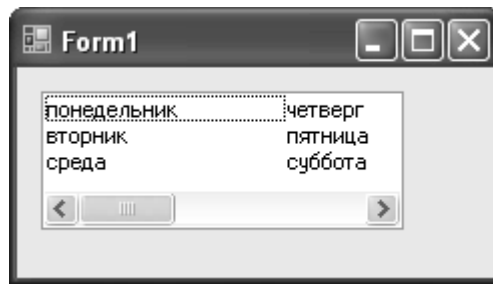
Компонент представляет собой список текстовых строк, из которых можно выбирать необходимые.

Свойства **ListBox**

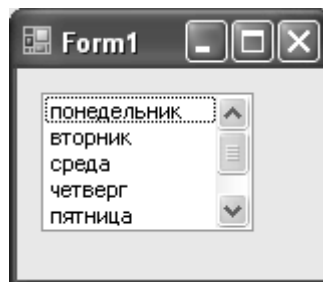
– **Items** – определяет коллекцию элементов, содержащихся в списке текстовых строк, позволяет их редактировать, добавлять и удалять. В окно редактора можно попасть, нажав в поле свойства кнопку с многоточием или щелкнув по кнопке со стрелкой на самом компоненте и выбрав в списке задач пункт **Правка элементов**.



– **MultiColumn** – вид отображения данных в столбцах. При выборе значения свойства **true** список отображается в виде нескольких колонок.



Если значение свойства **false**, то список элементов выводится в одну колонку.



- **ScrollAlwaysVisible** – постоянное отображение полосы прокрутки.

- **SelectionMode** – количество элементов, выбираемых в списке: **None** – ни одного, **One** – один, **MultiSimple** – несколько, **MultiExtended** – несколько с помощью клавиш на клавиатуре (**SHIFT**, **CTRL**, клавиши со стрелками).

- **Sorted** – сортировка списка по алфавиту.

Свойствами элементов можно управлять не только из окна **Свойства**, но в редакторе кода. Так свойства **ListBox.SelectedIndex**, **SelectedItem**, **Count** и многие другие не отображаются в окне **Свойства**.

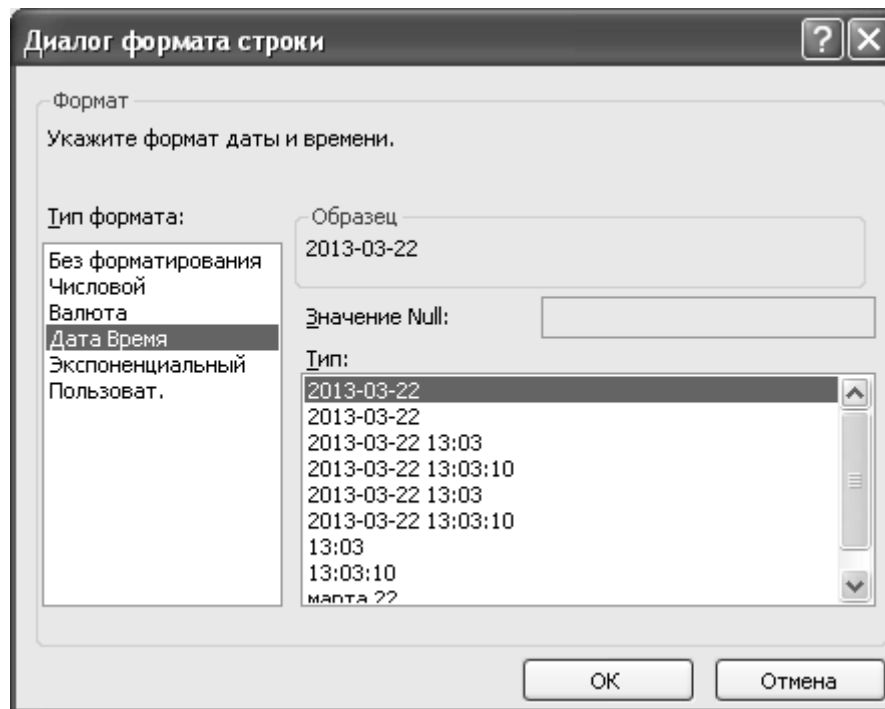
- **SelectedIndex** – номер выбранного элемента списка: **-1**, если не выбрано ни одного, **0** – выбран первый. При выборе нескольких элементов, свойство получает значение первого элемента из выделенных.

- **SelectedItem** – значение выбранного элемента в виде строки.

- **Text** – строка, выбранная в списке.

- **Count** – количество элементов в списке. При использовании этого свойства необходимо учитывать, что номер первого элемента равен 0.

– **FormatString** – способ отображения значений элементов списка. Целесообразно использовать данное свойство, если элементы списка являются данными типа, указанного в списке форматов. Окно диалога формата строки открывается путем нажатия кнопки с многоточием в поле данного свойства.



Основные методы компонента **ListBox** для работы с элементами списка:

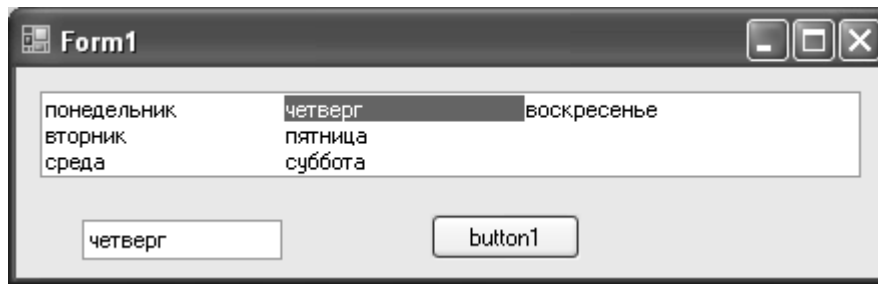
- **Add()** – добавить в конец списка;
- **Insert()** – вставить внутрь списка;
- **Clear()** – удалить все;
- **Remove()** – удалить нужную строку.

При выборе из списка другого элемента возникает событие **SelectedIndexChanged**. Создать обработчик этого события можно, выбрав его в окне свойств на вкладке **События** или выполнив двойной щелчок по компоненту в режиме **Конструктора**.

Например, мы хотим, чтобы при выборе в списке компонент отображался в текстовом поле. В обработчике события запишем:

```
textBox1->Text=listBox1->Text;
```

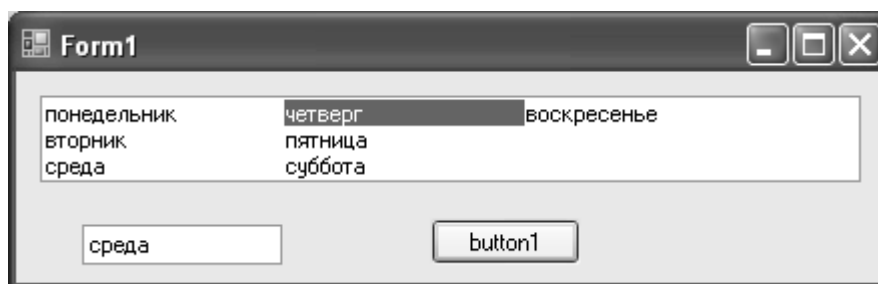
Результат представлен на рисунке:



Если требуется, чтобы в текстовом поле отображалось содержание конкретной строки списка, можно использовать следующую запись в обработчике события «Нажатие на кнопку»:

```
textBox1->Text=listBox1->Items[2]->ToString();
```

В результате в текстовом поле отобразится содержимое третьей строки списка. Важно помнить, что первая имеет индекс 0.



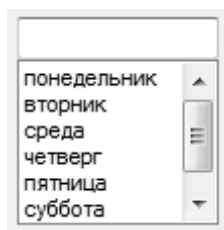
Компонент **ComboBox**

Данный компонент представляет собой текстовое поле с выпадающим списком. При выборе из списка какого-либо значения оно отображается в текстовом поле. В то же время в текстовое поле можно ввести значение, которое не содержится в списке.

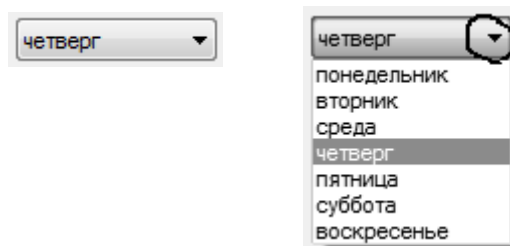
Большинство свойств и методов компонента **ComboBox** те же, что и у компонента **ListBox**. Однако присутствуют и отличия.

Свойства **ComboBox**

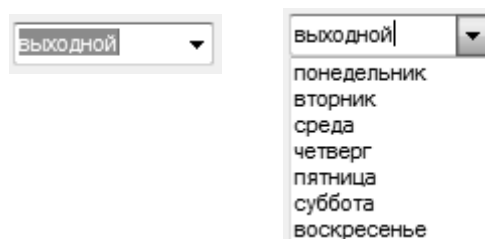
– **DropDownstyle** – вид отображения данных: **Simple** – список раскрыт всегда, в текстовое поле можно вводить данные, не входящие в список или выбирать их из списка.



– **DropDownList** – можно использовать только данные из списка, раскрыв его с помощью кнопки в текстовом поле, возможности редактирования нет.



– **DropDown** – можно использовать как данные из раскрывающегося списка, так и редактировать их, вводить в текстовое поле.



Используя свойства **AutoCompleteCustomSource**, **AutoCompleteMode** и **AutoCompleteSource**, можно обеспечить возможность автозаполнения поля со списком.

События компонента **ComboBox** очень сходны с событиями компонента **ListBox**. Также при выборе другого элемента из списка возникает событие **SelectedIndexChanged**. Но так как список может закрываться, то в данный момент происходит событие **DropDown-Closed**.

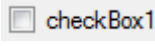
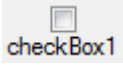
Добавлять и удалять строки из списка можно теми же методами, что и для компонента **ListBox**.

Компонент **CheckBox**

Компонент позволяет выбрать или отменить выбор какой-либо опции, то есть включить или отключить флажок. Можно использовать несколько флажков в группе. Они используются независимо друг от друга.

Свойства компонента **CheckBox**, отличные от рассмотренных ранее:

- **Appearance** – вид флажка: **Normal** – флажок, **Button** – кнопка. При выборе отображения флажка в виде кнопки включенный флажок выглядит как затемненная кнопка.

- **CheckAlign** – взаимное расположения флажка и текста с поясняющей надписью: **MiddleLeft** – , **TopCenter** – ,

- BottomCenter** – , **MiddleRight** –  и так далее.

- **Checked** – флажок включен, если равно **true**.

- **Checkstate** – компонент включен (**Checked**), выключен (**Unchecked**), просто выделен (**Indeterminate**).

- **ThreeState** – разрешить использование трех вышеописанных состояний.

Компонент **RadioButton**

Компонент используется как переключатель для выбора только одного из группы возможных вариантов. Остальные отключаются автоматически.

У данного компонента имеется только два варианта: включен или выключен, поэтому свойство **ThreeState** отсутствует. В остальных свойствах сходны со свойствами компонента **CheckBox**.

Компонент **GroupBox**

Данный компонент находится в окне **Панель элементов** в группе **Контейнеры**. Визуально представляет собой рамку с заголовком вокруг группы элементов. Использование **GroupBox** актуально, если используется группа зависимых переключателей, или необходимо выделить набор характеристик какого-либо объекта. Как и в случае с рассмотренным ранее компонентом **Panel**, все компоненты, разме-

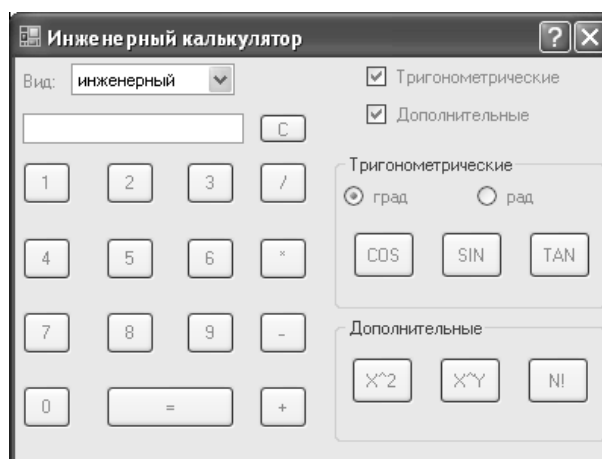
щенные в контейнере **GroupBox**, по умолчанию обладают некоторыми сходными с контейнером свойствами, могут одновременно перемещаться и обрабатываться.

Задание для выполнения.

Разработать приложение «Комбинированный калькулятор». Рекомендуемый интерфейс приложения представлен на рисунке (а – в режиме «обычный», б – в режиме «инженерный»).



а)



б)

Требования к приложению:

- переключение режимов работы производить при помощи поля со списком (компонент **ComboBox**);
- при помощи флажка «Отображать тригонометрические функции» управлять видимостью **GroupBox**, содержащей основные тригонометрические функции;
- иметь возможность выбора размера отображения (градусы или радианы) операнда при выполнении тригонометрических операций. Выбор должен осуществляться при помощи компонентов **RadioButton**.

Для успешного выполнения задания рекомендуется:

Определить свойство **Items** (содержание списка перечислений) компонента **ComboBox** на этапе разработки приложения. Для перехода в другой режим работы приложения (инженерный) анализировать

событие **OnChange** компонента.

При переходе в режим «**инженерный**» увеличить ширину формы, чтобы отобразить ее часть, используемую для расчета тригонометрических функций и соответственно уменьшить ее при переходе в режим «**обычный**».

Анализировать событие изменение флажка «Отображать тригонометрические функции» (**OnClick**) и управлять видимостью панели путем изменения свойства **Visible**.

В обработчиках события нажатие кнопки использовать для кнопок одного типа одинаковый обработчик событий:

цифровые кнопки

```
textBox1->Text=textBox1->Text+((Button^) sender)->Text;
```

кнопки со знаком операции

```
if (((Button^) sender)->Text=="+") Optn=1; и т.д.
```

Следующие переменные объявить после строки

```
using namespace System::Drawing;
```

```
double Rad=180/Math::PI;
```

```
double Op1,Op2;
```

```
int Optn;//0-нет;1-слож;2-вычит;3-умн;4- дел
```

Код программы для проверки

Файл Form1.h

```
#pragma once
```

```
namespace zadanie {
```

```
using namespace System;
```

```
using namespace System::ComponentModel;
```

```
using namespace System::Collections;
```

```
using namespace System::Windows::Forms;
```

```
using namespace System::Data;
```

```
using namespace System::Drawing;
```

```
double Rad=180/Math::PI;
```

```
double Op1,Op2;
```

```
int Optn;//0-нет;1-слож;2-вычит;3-умн;4- дел
```

. . .

Обработчик события **DropDownClosed** текстового поля с выпадающим списком **comboBox1**, изменяющий размеры формы:

```
{  
    if (comboBox1->Text=="обычный")  
        Width=200;  
    else Width=400;  
}
```

Обработчик события формы **Load**, устанавливающий ширину формы при ее загрузке:

```
{  
    comboBox1->SelectedIndex=0;  
    Width=200;  
}
```

Обработчик события **SelectedIndexChanged** текстового поля с выпадающим списком **comboBox1**, изменяющий текст заголовка формы:

```
{  
if (comboBox1->SelectedIndex==1)  
    this->Text="Инженерный калькулятор";  
else  
    this->Text="Калькулятор";  
}
```

Обработчик события **CheckedChanged** флажка **checkBox1**, устанавливающий видимость контейнера с тригонометрическими функциями в зависимости от включения данного флажка:

```
{ groupBox1->Visible=checkBox1->Checked; }
```

Обработчик события **CheckedChanged** флажка **checkBox2**, устанавливающий видимость контейнера с дополнительными функциями в зависимости от включения данного флажка:

```
{ groupBox2->Visible=checkBox2->Checked; }
```

Обработчик события «Нажатие на кнопку **С**» (очистка), выполняющий очистку текстового поля:

```
{ textBox1->Clear(); }
```

Обработчик события «Нажатие на кнопку с **цифрой**». Достаточно написать этот обработчик для одной из кнопок с изображением цифры, а потом выделить все остальные и на вкладке **События** окна **Свойства** выбрать данное событие в списке событий **Click**:

```
{  
textBox1->Text=textBox1->Text+((Button^) sender)->  
Text;  
}
```

Обработчик события «Нажатие на кнопку со **знаком операции**(+, -, *, /)». Также как в предыдущем случае можно написать для одной кнопки и выбрать для всех остальных:

```
{  
if (textBox1->Text!="")  
Op1=Convert::ToDouble(textBox1->Text);  
else Op1=0;  
if (((Button^) sender)->Text=="+") Op1n=1;  
if (((Button^) sender)->Text=="-") Op1n=2;  
if (((Button^) sender)->Text=="*") Op1n=3;  
if (((Button^) sender)->Text=="/") Op1n=4;  
textBox1->Text="";  
}
```

Обработчик события «Нажатие на кнопку со **знаком =**»:

```
{double rez,
  Op2=Convert::ToDouble(textBox1->Text);

  switch(Optn)
  {
    case 1: rez=Op1+Op2;break;
    case 2: rez=Op1-Op2;break;
    case 3: rez=Op1*Op2;break;
    case 4: rez=Op1/Op2;break;
  }
  textBox1->Text=rez.ToString();
  Optn=0;
}
```

Обработчик события «Нажатие на кнопку со **тригонометрическими функциями (SIN, COS, TAN)**»:

```
{double rez;
if (textBox1->Text!="")
    Op1=Convert::ToDouble(textBox1->Text);
else Op1=0;
if (radioButton1->Checked==true)
    Op1=Op1/Rad;
if (((Button^) sender)->Text=="SIN")
    rez=Math::Sin(Op1);
if (((Button^) sender)->Text=="COS")
    rez=Math::Cos(Op1);
if (((Button^) sender)->Text=="TAN")
    rez=Math::Tan(Op1);

textBox1->Text=rez.ToString();
}
```

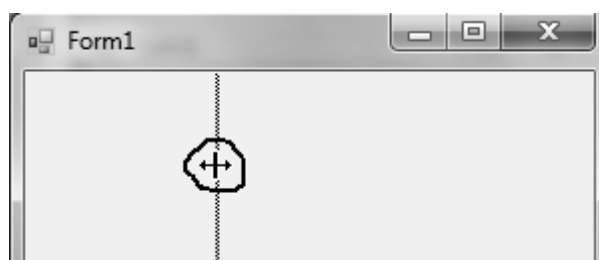

Глава 7

Компоненты, обеспечивающие диалог с пользователем

7.1. Компоненты SplitContainer, MenuStrip, ContextMenuStrip, RichTextBox, OpenFileDialog, SaveFileDialog, ColorDialog, FontDialog

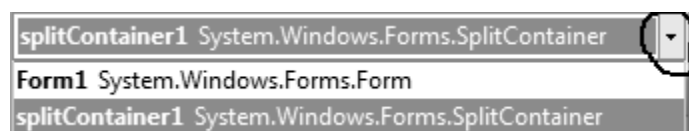
Компонент SplitContainer

Данный компонент, так же как и компоненты **GroupBox** и **Panel** располагается в окне **Панель элементов** в группе **Контейнеры**. Однако в отличие от данных компонентов его можно использовать для разделения контейнера на две части. Размеры составных частей данного контейнера можно изменять с помощью указателя мыши, наведенного на линию разделения.



С помощью компонента **SplitContainer** можно разделить на части **GroupBox**, **Panel**, форму и другие компоненты-контейнеры.

Чтобы получить доступ к свойствам данного компонента, проще выделить его с помощью окна **Свойства**, воспользовавшись выпадающим списком вверху окна:

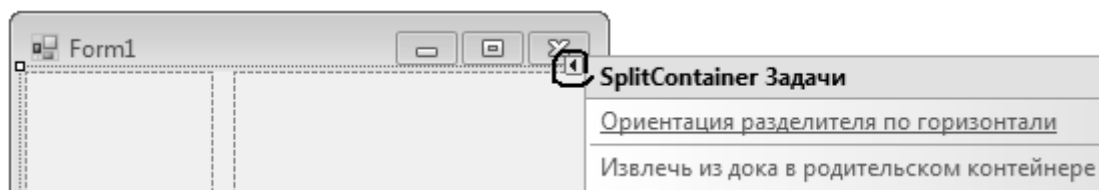


Некоторые свойства компонента:

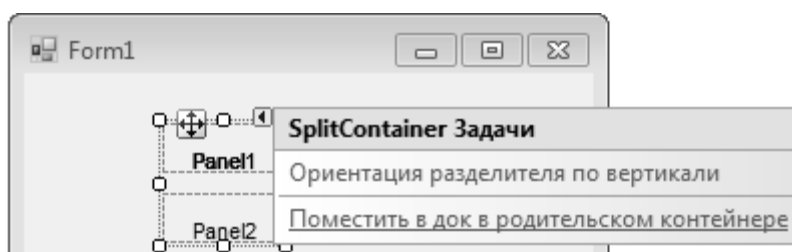
- **FixedPanel** – сохранение размера одной из панелей компонента (None – никакая, Panel1 – первая панель, Panel2 – вторая панель), если он сам изменяет размеры.

- **Orientation** – вертикальное (**Vertical**) или горизонтальное (**Horizontal**) разделение контейнера.

К этому свойству можно получить доступ и из окна формы, нажав кнопку со стрелкой в правом верхнем углу компонента.



Там же присутствует функция, которая может свернуть **SplitContainer** или растянуть его на весь контейнер. Данная возможность особенно полезна, если вы хотите удалить **SplitContainer** с формы. В растянутом положении это сделать трудно.



– **IsSplitterFixed** – фиксация разделителя при установке значения свойства **true**.

– **SplitterIncrement** – точность с которой можно перемещать разделительную линию.

– **SplitterWidth** – ширина разделительной линии.

Компонент MenuStrip

Компонент **MenuStrip** находится в окне **Панель элементов** в группе **Меню и панели инструментов** и используется для создания главного меню приложения.

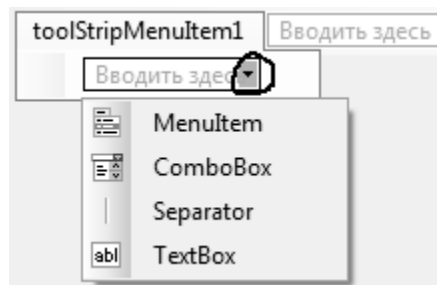
Чтобы создать главное меню нужно выполнить ряд шагов. Сначала необходимо поместить значок на форму. Имя данного меню автоматически присвоится свойству формы **MainMenuStrip**.

В режиме **Конструктора** значок компонента расположится на серой полосе внизу окна формы.

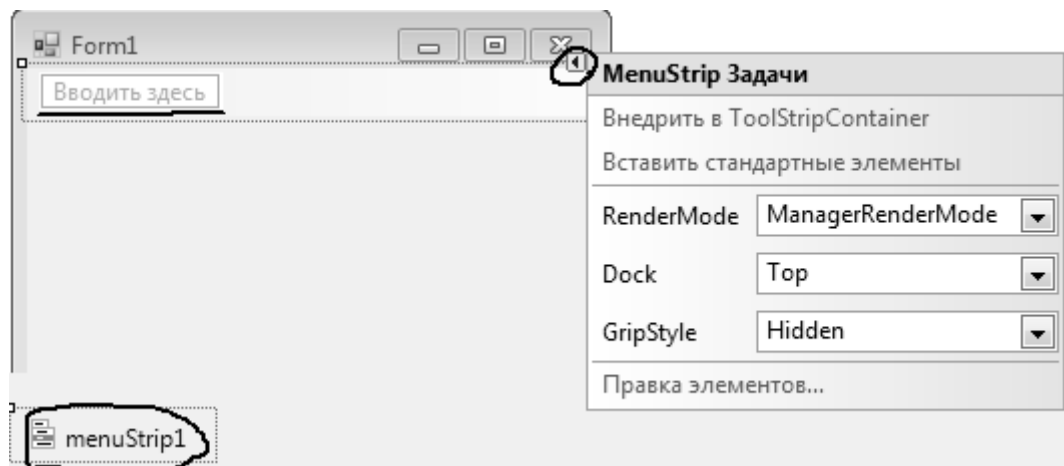
В верхней части формы появится поле, в которое можно вводить текст меню.

После заполнения данного поля появятся поля внизу и справа,

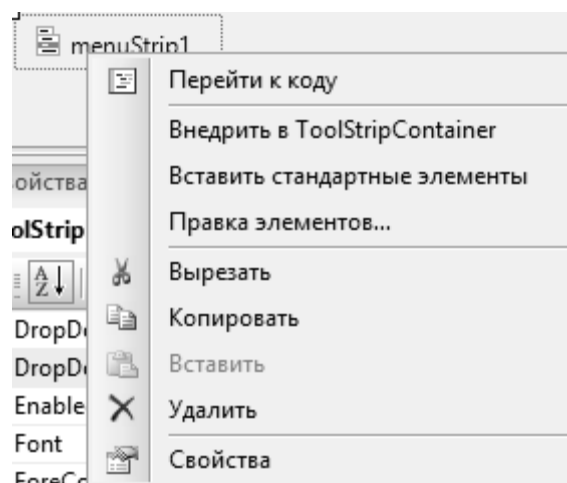
заполнить которые можно как текстом, так и выбрав из выпадающего списка нужный вариант заполнения.



Если нажать кнопку в правом верхнем углу компонента, получим доступ к дополнительным возможностям.



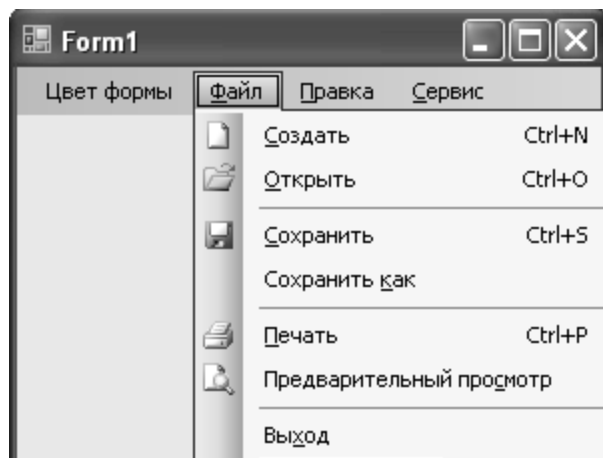
Те же функции обеспечивает и контекстное меню компонента.



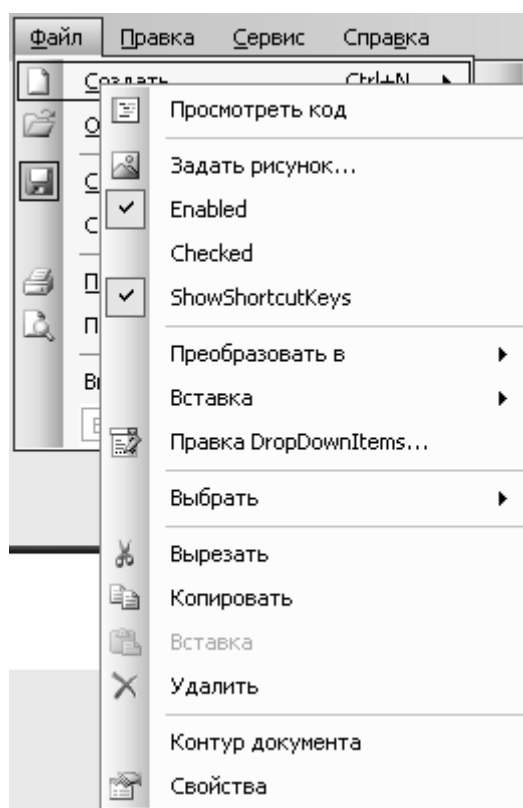
Внедрить в **ToolStripContainer** – помещает меню в контейнер, при этом меню располагается в области контейнера, а не вдоль всей

верхней границы формы.

Вставить стандартные элементы – наряду с созданными пунктами меню добавляются **Файл, Правка, Сервис, Справка**.

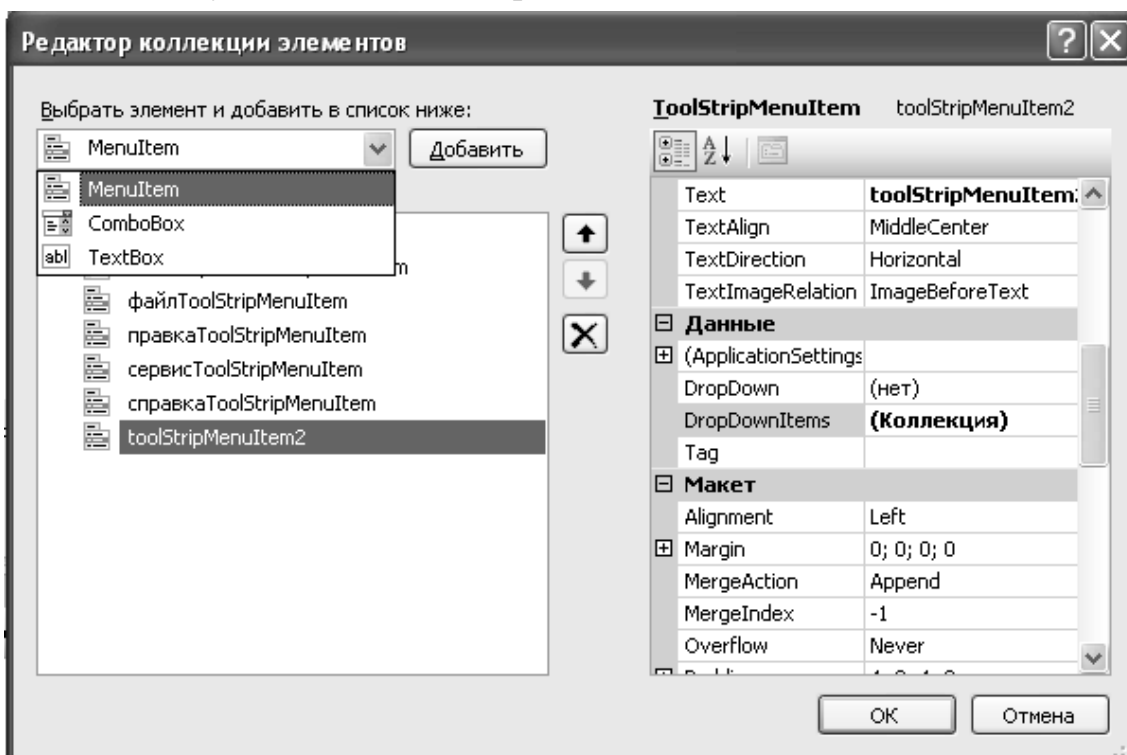


Пункты меню затем можно корректировать: перетаскивать их с помощью кнопки мыши, удалять и преобразовывать в текстовое поле, поле со списком, разделитель с помощью контекстного меню каждого из пунктов.



Правка элементов... – редактирование коллекции элементов. В

открывшемся диалоговом окне можно удалять, перемещать и добавлять новые пункты меню и настраивать их свойства.



За пунктами меню закрепляются обработчики событий, которые создаются двойным щелчком левой кнопки мыши по нужному пункту или в окне **Свойства** на вкладке **События**.

Некоторые свойства компонента **MenuStrip** устанавливаются для компонента в целом, а некоторые только для конкретного пункта меню при его выделении:

- **Items** – открывается редактор коллекции элементов, также как по нажатию пункта контекстного меню **Правка элементов...**

- **LayoutStyle** – ориентация меню: **StackWithOverflow** – автоматически, **HorizontalStackWithOverflow** – горизонтально, **VerticalStackWithOverflow** – вертикально по центру, **Table** – вертикально по левому краю, **Flow** – с переносом элементов на другую строку.

- **BackgroundImage** – фоновое изображение всей полосы меню или отдельного пункта.

- **BackColor** – цвет фона.

- **BackgroundImageLayout** – вариант размещения изображения на компоненте.

– **ShortcutKeys** – «горячие» клавиши. **ShowShortcutKeys** должно быть **true**.

– **CheckOnClick** – выбор пункта меню отмечается флажком.

Компонент ContextMenuStrip

Компонент **ContextMenuStrip** находится в окне **Панель элементов** в группе **Меню и панели инструментов** и используется для создания контекстного меню другого компонента. Меню данного вида в работающем приложении появляется при щелчке правой кнопкой мыши по форме или другому компоненту, к которому оно прикреплено.

Также как и компонент **MenuStrip** значок данного компонента располагается на серой полосе внизу окна формы. Чтобы присоединить данное меню к какому-либо компоненту надо выбрать в свойстве **ContextMenuStrip** данного компонента имя контекстного меню. Свойства и процесс создания данного компонента такие же, как и у компонента **MenuStrip**.

Компонент RichTextBox

Компонент **RichTextBox** располагается в окне **Панель элементов** в группе **Стандартные элементы управления**.

С помощью данного компонента можно работать с форматированным текстом. Это его главное отличие от компонента **TextBox**. В остальном они имеют сходные свойства, события и методы, которые будут рассмотрены ниже при выполнении практического примера.

Рассматриваемые далее компоненты **OpenFileDialog**, **SaveFileDialog**, **ColorDialog**, **FontDialog** находятся в окне **Панель элементов** в группе **Диалоговые окна**.

Компонент OpenFileDialog

Компонент **OpenFileDialog** используется для открытия файлов при помощи диалогового окна.

Как и все остальные компоненты из этой группы, при помещении на форму компонент **OpenFileDialog** размещается на серой полосе внизу окна формы и в работающем приложении не отображается.

Общим для всех компонентов диалога является также то, что для каждого из них отображение диалогового окна происходит с помощью метода **ShowDialog()**.

Некоторые свойства компонента **OpenFileDialog**:

- **AddExtension** – автоматическое добавление расширений файлов.
- **CheckFileExists** – проверка существования выбранного файла.
- **CheckPathExists** – проверка правильности пути к файлу.
- **DefaultExt** – расширение файла, которое добавляется автоматически.
- **FileName** – имя файла, выбранного в диалоговом окне.
- **Filter** – указываются расширения файлов, доступных в работающем приложении при сохранении или отображении файлов. Данные расширения отображаются в поле **Тип файлов** диалогового окна.
- **Multiselect** – выбор нескольких файлов.
- **ShowHelp** – отображение кнопки **Справка**.
- **ShowReadOnly** – отображение флажка для файлов, доступных только для чтения.
- **Title** – строка заголовка диалогового окна.
- **ValidateNames** – проверка допустимости использования имен файлов.

Компонент **SaveFileDialog**

Компонент **SaveFileDialog** используется для сохранения файла с помощью диалогового окна, позволяющего выбрать место, имя и расширение сохраняемого файла.

Свойства данного компонента в большинстве те же, что у рассмотренного ранее компонента **OpenFileDialog**.

Однако между ними присутствуют и некоторые отличия. Так у компонента **SaveFileDialog** отсутствует свойство **Multiselect**, но имеется свойство **OverwritePrompt** – отображение запроса на перезапись имеющегося файла. При этом необходима установка значения **true** свойству **ValidateNames**.

Компонент **ColorDialog**

Компонент **ColorDialog** предназначен для выбора цвета из палитры с помощью диалогового окна.

Некоторые свойства компонента **ColorDialog**:

- **AllowFullOpen** – возможность определения собственных цветов.
- **AnyColor** – отображение всех доступных цветов в базовом наборе.
- **Color** – выбранный в диалоговом окне цвет.
- **SolidColorOnly** – отображение только чистых цветов.

Компонент **FontDialog**

Компонент **FontDialog** используется для выбора и настройки шрифтов при помощи диалогового окна.

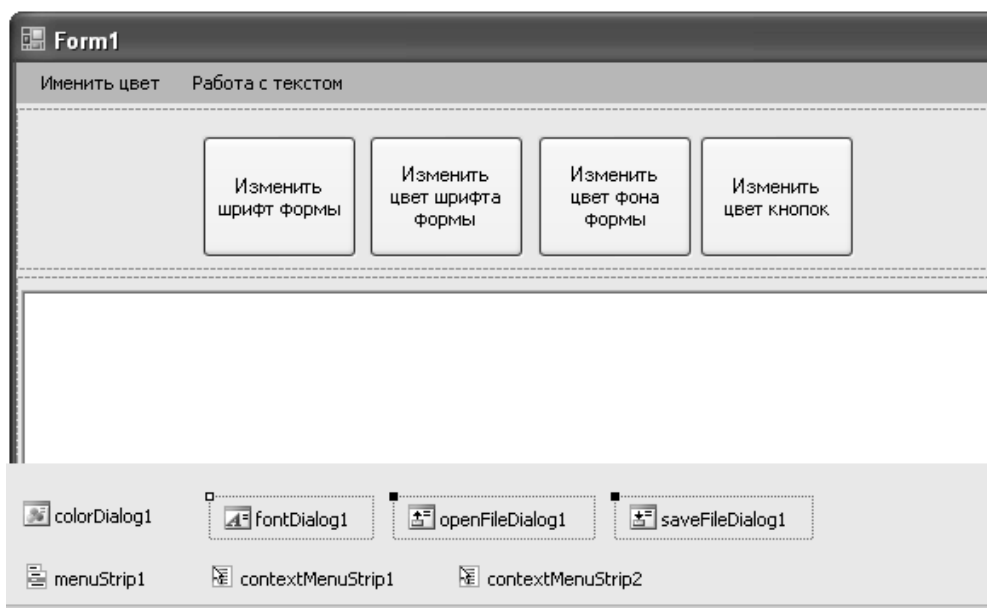
Некоторые свойства компонента **FontDialog**:

- **Font** – название и размер выбранного шрифта.
- **Color** – цвет выбранного шрифта.
- **ShowColor** – возможность выбора цвета шрифта в диалоговом окне.
- **ShowEffect** – выбор варианта видоизменения шрифта, например подчеркивание.

Задание для выполнения 1.

Разработать приложение для исследования возможностей компонентов диалога.

Рекомендуемый интерфейс приложения в режиме конструктора представлен на рисунке.



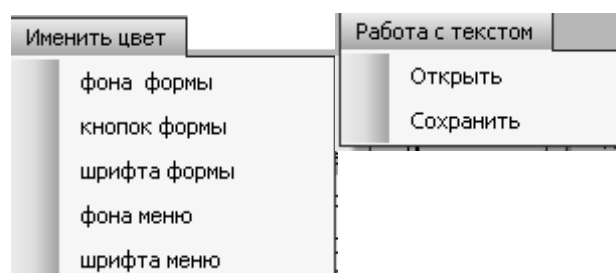
Для получения результата нужно выполнить следующие шаги:

- Добавить на форму компонент **splitContainer**. Настроить его горизонтальную ориентацию.
- На верхнюю панель добавить кнопки.
- На нижнюю панель добавить компонент **richTextBox**.
- Добавить компоненты диалога в соответствии с рисунком.
- Установить свойство **Filter** компонентов **openFileDialog** и **saveFileDialog**, чтобы в поле **Тип файлов** отображались варианты выбора текстовых файлов, файлов с расширением **.rtf** и всех файлов.

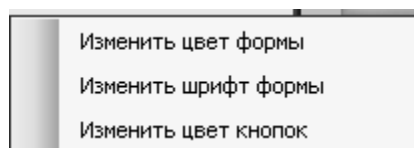
Text Files(*.txt)|*.txt|RTF Files)|*.rtf|All files(*.*)|*.*

- Добавить главное меню и два контекстных.

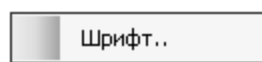
Главное меню



Контекстное к форме



Контекстное к richTextBox



– Одно контекстное меню присоединить к форме, другое к компоненту **richTextBox**.

Обработчики событий:

Для изменения **шрифта** формы и одновременно шрифта пунктов меню(чтобы были как у формы)

```
fontDialog1->ShowDialog();  
Font=fontDialog1->Font;  
menuStrip1->Font=Font;
```

Для изменения **цвета шрифта** формы и одновременно шрифта пунктов меню(чтобы были как у формы)

```
colorDialog1->ShowDialog();  
ForeColor=colorDialog1->Color;  
menuStrip1->ForeColor=ForeColor;
```

Для изменения **цвета фона** формы

```
colorDialog1->ShowDialog();  
BackColor=colorDialog1->Color;
```

Для изменения **цвета кнопок**

```
colorDialog1->ShowDialog();  
button1->BackColor=colorDialog1->Color;  
button2->BackColor=colorDialog1->Color;  
button3->BackColor=colorDialog1->Color;  
button4->BackColor=colorDialog1->Color;
```


Для **открытия** файла и **отображения** его в компоненте **richTextBox**

```
openFileDialog1->ShowDialog();  
richTextBox1->LoadFile(openFileDialog1->FileName);
```

Для **сохранения** файла, отображенного в компоненте **richTextBox**

```
saveFileDialog1->ShowDialog();  
richTextBox1->SaveFile(saveFileDialog1->FileName);
```

Для изменения **шрифта текста** в компоненте **richTextBox**

```
fontDialog1->ShowDialog();  
richTextBox1->Font=fontDialog1->Font;
```

Задание для выполнения 2.

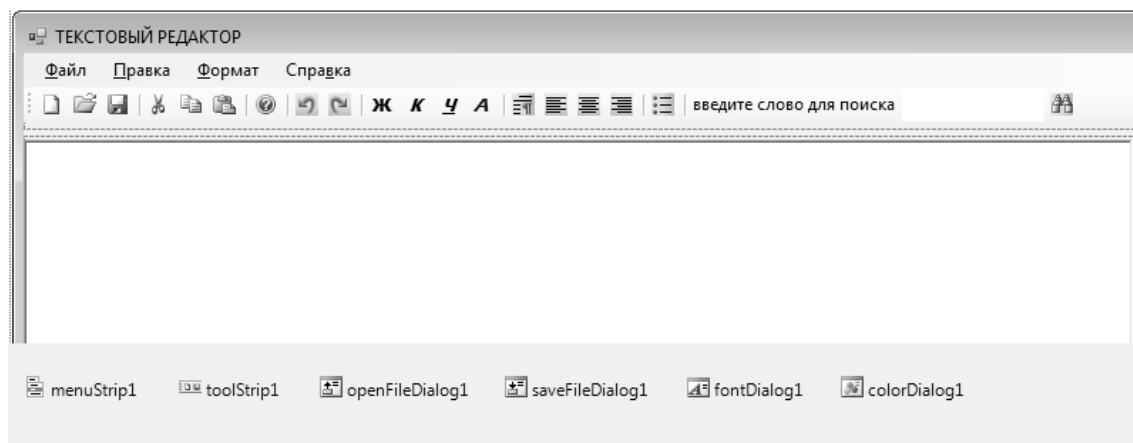
Разработать простейший текстовый редактор для работы с текстовыми файлами на диске. Дополнить разработанное при выполнении **Задания 1** приложение, обеспечив выполнение следующих функций:

- При запуске приложения запускается вторая форма, которая закрывается при выполнении щелчка левой кнопкой мыши по ней.
- Для работы с файлами и выполнения форматирования текста используется главное меню и кнопки на панели инструментов.
- По желанию можно использовать стандартные кнопки или использовать собственные кнопки с картинками (для создания собственных изображений для кнопок можно использовать любой графический редактор, создав и сохранив картинку нужного размера),
- При открытии файла его имя отображается в заголовке формы.
- Для выполнения одинаковых действий с помощью меню и кнопок обработчик события прописывается для одного из компонентов, для второго – выбираются из списка.
- Обеспечивается возможность поиска текста в файле и получения справки в диалоговом окне.
- Присутствуют всплывающие подсказки, изменение вида кур-

сора при наведении на кнопки.

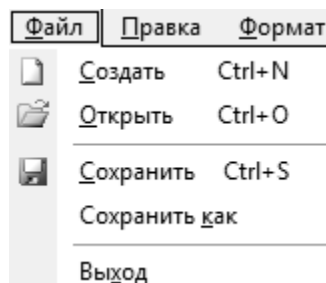
– Дополнительно можно добавить диалоговое окно для изменения параметров абзаца (подобное окну изменения шрифта).

Рекомендуемый интерфейс первой формы приложения в режиме конструктора представлен на рисунке.

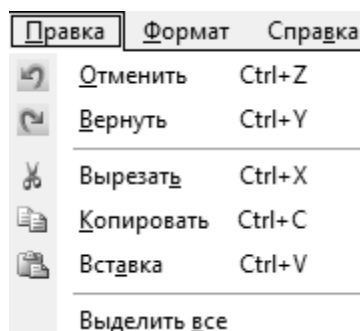


Главное меню

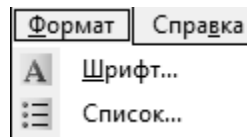
Пункт Файл



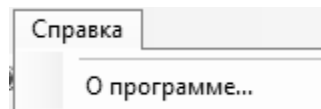
Пункт Правка



Пункт Формат



Пункт Справка



Вариант формы-заставки

Заставку оформить, добавив вторую форму **Form2.h[Конструктор]** и разместив на ней текстовую надпись или рисунок по желанию. Можно также обеспечить развертывание заставки на весь экран с помощью свойства формы **WindowState**, установив его значение **Maximized**.

Открываться форма-заставка должна при запуске приложения.



Обработчик события **Shown** для **Form1**, в результате выполнения которого при первой загрузке первой формы отображается вторая форма-заставка:

```
Form2^ f2=gcnew Form2();  
f2->Show();
```

Обработчик события **Click** для **Form2**, в результате выполнения которого форма-заставка закрывается при щелчке по ней левой кнопкой мыши:

```
Close();
```

Рекомендации для выполнения задания:

- изначально компонент **richTextBox1** невидим (свойство **Visible** определить на этапе разработки приложения);

- пункт меню **Файл** и кнопка **Создать**: отображается и очищается компонент **richTextBox1**

```
richTextBox1->Visible=true;  
richTextBox1->Clear();
```

- пункт меню **Файл** и кнопка **Открыть**: отображается **richTextBox1**, в компоненте **richTextBox1** открывается выбранный с помощью окна диалога файл и изменяется заголовок формы (отображается имя файла **openFileDialog1->FileName**)

```
richTextBox1->Visible=true;  
openFileDialog1->ShowDialog();  
richTextBox1->LoadFile(openFileDialog1->  
>FileName);  
this->Text=openFileDialog1->FileName;
```

- пункт меню **Файл** и кнопка **Сохранить**: сохраняется открытый ранее и измененный файл, имя файла – в заголовке формы

```
richTextBox1->Visible=true;  
richTextBox1->SaveFile(this->Text);
```

- пункт меню **Файл** и кнопка **Сохранить как**: отображается **richTextBox1**, файл сохраняется с помощью окна диалога.

```
richTextBox1->Visible=true;  
saveFileDialog1->ShowDialog();
```

```
richTextBox1->SaveFile(saveFileDialog1->FileName);
```

– пункт меню **Файл** и кнопка **Выход**: закрыть приложение
`Close();`

– пункт меню **Правка** и кнопка **Отменить**
`richTextBox1->Undo();`

– пункт меню **Правка** и кнопка **Вернуть**
`richTextBox1-> Redo();`

– пункт меню **Правка** и кнопки **Вырезать** (`richTextBox1-> Cut();`), **Копировать** (`richTextBox1-> Copy();`), **Вставить** (`richTextBox1-> Paste();`).

– пункт меню **Правка** и кнопка **Выделить все**
`richTextBox1-> SelectAll();`

– пункт меню **Формат** и кнопка **Шрифт**, для изменения шрифта в выделенном фрагменте с помощью окна диалога

```
fontDialog1->ShowDialog();  
richTextBox1->SelectionFont=fontDialog1->Font;
```

– пункт меню **Формат** и кнопка **Список**: абзац, где находится курсор, или выделенные абзацы маркируются, при повторном нажатии – обычный текст

```
if (richTextBox1->SelectionBullet == true)  
{ richTextBox1->SelectionBullet = false; }  
  
else  
{ richTextBox1->SelectionBullet = true; }
```

– пункт меню **Справка/ О программе** и кнопка **Справка**: выводится справочное сообщение

```
MessageBox::Show("Программа предназначена  
\ndля работы с текстовыми файлами\n в формате.rtf  
и .txt\n", "СПРАВКА",MessageBoxButtons::OK,  
MessageBoxIcon::Information);
```

Дополнительно кнопки на панели инструментов:

– выравнивание абзаца: по левому краю,

```
richTextBox1->  
SelectionAlignment=HorizontalAlignment::Left;
```

по центру,

```
richTextBox1->  
SelectionAlignment=HorizontalAlignment::Center;
```

по правому краю

```
richTextBox1->  
SelectionAlignment=HorizontalAlignment::Right;
```

– жирный шрифт **Bold**

```
// проверка того, что текст выделен  
if ( richTextBox1->SelectionFont != nullptr )  
{  
//определяется формат текста с конкретными  
//характеристиками  
System::Drawing::Font^ curF = richTextBox1->  
>SelectionFont;  
  
//определяется стиль текста  
System::Drawing::FontStyle newFSt;  
//проверяется, ли выделенный текст полужирным  
if ( richTextBox1->SelectionFont->Bold ==  
true )  
  
{
```

```

//устанавливается стандартный стиль текста
newFSt = FontStyle::Regular;}
    else
    {
//устанавливается полужирный стиль текста
newFSt = FontStyle::Bold;
}

//установленный стиль текста применяется к
//выделенному
    richTextBox1->SelectionFont = gcnw
System::Drawing::Font( curF->FontFamily,curF-
>Size,newFSt );
}
-    курсив – то же только вместо Bold - Italic
-    подчеркивание – то же только вместо Bold – Underline
-    отступ абзаца
richTextBox1->SelectionIndent = 20;

-    поиск текста по строке, введенной в поле с учетом реги-
стра и выделение его синим цветом, подчеркиванием и увеличением
размера символов

    String ^text;
    text=toolStripTextBox1->Text;
richTextBox1->Find( text,
RichTextBoxFinds::MatchCase );

richTextBox1->SelectionFont = gcnw
System::Drawing::Font("Arial", 14,  FontStyle::
Underline);

richTextBox1->SelectionColor = Color::Blue;

```

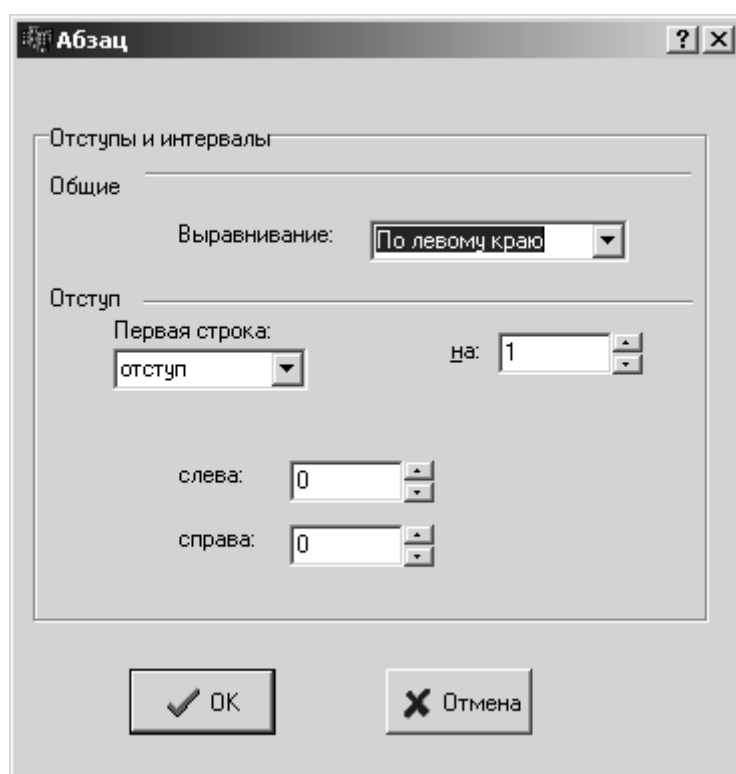
В результате искомое слово изменит свой цвет, шрифт и будет подчеркнуто:



– изменение цвета шрифта выделенного текста с помощью окна диалога

```
colorDialog1->ShowDialog();
richTextBox1->SelectionColor = colorDialog1-
>Color;
```

Предлагаемый интерфейс диалогового окна для изменения параметров абзаца приводится на рисунке.



Данная функция текстового редактора предлагается в качестве дополнительной. Ее предлагается реализовать самостоятельно, добавив в пункт меню **Формат** необходимую строку **Абзац...**, и, составив обработчик события, воспользовавшись приведенными выше рекомендациями для кнопок выравнивания текста, установки абзацного отступа.

Глава 8

Обеспечение работы с файлами и Web-страницами

8.1. Программирование операций файлового ввода-вывода в Windows-приложениях

Одним из способов организации работы с файлами является использование класса **File**, который обладает методами, позволяющими создавать, копировать, удалять, перемещать, открывать файлы и т.д.

Рассмотрим некоторые методы данного класса, которые мы будем использовать при выполнении практических заданий:

- **AppendAllText** – в файл добавляется строка. Если файла нет, то он создается.
- **Delete** – удаление файла.
- **ReadAllText** – открытие файла, считывание его содержимого в одну строку и закрытие файла.
- **ReadAllLines** – открытие файла, считывание его содержимого в массив строк и закрытие файла.
- **WriteAllLines** – создание файла, запись в него строки и закрытие файла.

Задание для выполнения 1.

Разработать приложение для исследования возможностей файлового ввода-вывода.

Рекомендуемый интерфейс приложения приведен на рисунке.



Рекомендации для выполнения задания.

- Для записи текста в файл использовать компонент **TextBox1** и кнопка «**Запись в файл**».
- Свойство **Multiline** компонента **TextBox1** установить **true**.
- Текст вводится в компонент **TextBox1**, по нажатию на кнопку «**Запись в файл**» – записывается в файл.
- Для чтения текста из файла используется компонент **TextBox2** и кнопка «**Чтение из файла**».
- Для ввода/вывода файлов применяются методы из класса **File**, поэтому при выполнении всех заданий этой главы необходимо использовать

```
using namespace System::IO;
```

Для работы с файлами определить две функции: **SvTxt()** – для записи в файл и **RdTxt()** – для чтения их файла. Определение функций необходимо добавить в **Form1.h** после строк

```
public ref class Form1 : public System::Windows::Forms::Form
{
void SvTxt(String ^F, TextBox ^t)
{
//удаление файла
File::Delete(F);

//создание массива строк CLR с именем tmass
// размерностью t->Lines->Length
array<String^>^ tmass=gcnw array<String^>(t-
>Lines->Length);

//заполнение массива строками из компонента textBox1
tmass=textBox1->Lines;
// создание файла, запись в него массива строк и закрытие файла
File::WriteAllLines(F,tmass);
}
```

```

void RdTxt(String ^F, TextBox ^t)
{
//установка многострочного режима компонента TextBox
    t->Multiline=true;

//очистка компонента TextBox
    t->Clear();

// открытие файла, считывание его содержимого в массив строк
// и закрытие файла
    t->Lines=File::ReadAllLines(F);
}

```

Обработчик события «Нажатие на кнопку **Запись в файл**»:

```

//в записи пути к файлу пишутся две косые черты
//файл будет создан и сохранится на диске d под именем 20.txt
SvTxt("d:\\20.txt",this->textBox1);

```

Обработчик события «Нажатие на кнопку **Чтение из файла**»:

```

//открываемый файл должен присутствовать на диске
RdTxt("d:\\20.txt",this->textBox2);

```

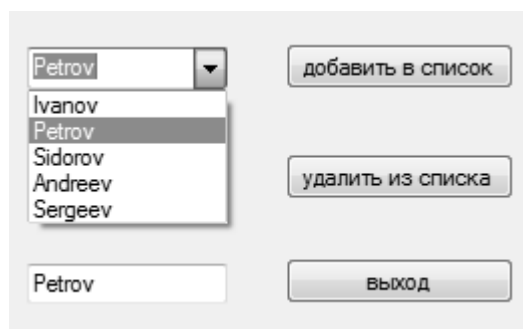
8.2. Загрузка текстовых строк из файла в компонент **ComboBox**

Для хранения строк, определяющих содержимое компонента **ComboBox**, также можно использовать файл. Если таких строк не очень много, то можно обойтись без использования массивов, а соединить их всех в одну. При этом после каждой записываемой строки нужно использовать какой-либо разделяющий символ. Например, наклонную черту /.

Задание для выполнения 2.

Разработать приложение, в котором содержимое компонента **ComboBox** определяется содержимым файла.

Рекомендуемый интерфейс приложения представлен на рисунке



Рекомендации для выполнения задания.

– В папке приложения, где хранится файл формы **Form1.h**, создать текстовый файл **a.txt**. Записать туда латинским шрифтом любые фамилии в следующем виде: **Ivanov/Petrov/Sidorov/Andreev/Sergeev/**

После окончания разработки приложения и создания исполняемого файла **.exe** при конфигурации **Release** текстовый файл нужно скопировать в папку **Release**, или другую, где будет находиться исполняемый файл.

– Сохранить файл.

– При загрузке приложения (событие формы **Load**) должен читаться созданный ранее текстовый файл **a.txt**, в котором находятся сведения о предыдущей работе приложения. Содержимым файла заполняется компонент **ComboBox**.

– При выборе строки в компоненте **ComboBox** она должна отображаться в компоненте **TextBox**.

– Содержимое **ComboBox** можно редактировать: добавлять новые строки, набранные в **TextBox**, удалять выбранные, используя соответствующие кнопки.

– Для работы с файлами нужно использовать функцию для загрузки текстовых строк из обыкновенного текстового файла, подготовленного программой **Блокнот** или **Word**.

– Применить данную функцию для загрузки строк и вывести отмеченную строку в компонент **TextBox**.

– Для изменения содержимого файла после редактирования **ComboBox** также использовать функцию

– Поместить созданные функции там же, где и функции в *Зада-
нии для выполнения 1*.

Функция для чтения строки из файла, разбиения ее на отдель-
ные части по разделителю /, отображения строк в компоненте **Com-
boBox**:

```
void RStrF(String ^F, ComboBox ^cb)
{
// открытие файла, считывание его содержимого в одну строку ^sb и
заккрытие файла
    String ^st, ^sb=File::ReadAllText(F);

//очистка содержимого выпадающего списка ComboBox
    cb->Items->Clear();

// разбиение считанной строки на отдельные части по разделителю /
    while (sb->Length>0)
    {
//обнаружение первого разделителя /
        int i=sb->IndexOf("/");

// если разделителя нет – выход из цикла
        if (i==-1) break;

//извлечение подстроки расположенной после разделителя
        st=sb->Substring(0,i);

//добавление извлеченной подстроки в компонент ComboBox
        cb->Items->Add(st);

//определение оставшейся строки
        sb=sb->Substring(i+1, sb->Length-st->Length-1);
    }
    return; }
```

Функция для сохранения содержимого компонента **ComboBox** в файле.

```
void SStrF(String ^F, ComboBox ^cb)
{
String ^sa,      //i-ая строка ComboBox
^sb;             //объединенная строка ComboBox
//определение количества строк в ComboBox
int j=cb->Items->Count;

//удаление имеющегося файла, где хранится содержимое ComboBox
File::Delete(F);

for(int i=0;i<j;i++)
{
//считывание i-й строки из компонента ComboBox
sa=cb->Items[i]->ToString();

//присоединение i-й строки и разделителя / к объединенной строке
sb+=sa->Concat(sa, "/" );
}
//добавление в файл объединенной строки и закрытие файла
File::AppendAllText(F, sb);
return;
}
```

Обработчик события **Load** формы **Form1** :

```
//считывание из файла a1.txt содержимого списка
RStrF("a1.txt", this->comboBox1);
```

Обработчик события «Нажатие на кнопку **Добавить в список**»:

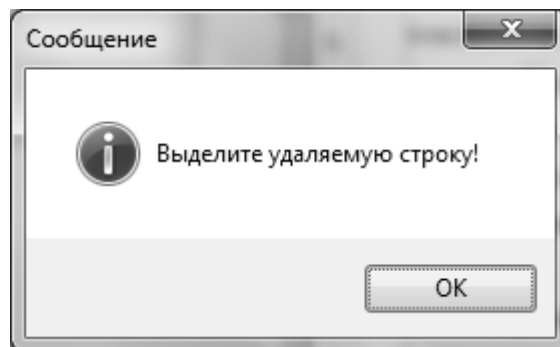
```
//в выпадающий список ComboBox добавляется строка
//из текстового поля TextBox
comboBox1->Items->Add(textBox3->Text);
```

Обработчик события «Нажатие на кнопку **Удалить из списка**»:

```
//если в ComboBox не выделено ни одной строки – выводится
//сообщение об этом
if (comboBox1->SelectedIndex==-1)
{
    MessageBox::Show("Выделите удаляемую строку!",
        "Сообщение", MessageBoxButtons::OK,
        MessageBoxIcon::Asterisk);
}
//из выпадающего списка ComboBox удаляется выделенная строка
comboBox1->Items->Remove (comboBox1->SelectedItem);

// очистка содержимого текстового поля
textBox3->Clear();
```

Окно сообщения при отсутствии выделенных строк



Обработчик события «Нажатие на кнопку **Выход**»:

```
//содержимое выпадающего списка сохраняется в файл a1.txt
SStrF ("a1.txt", this->comboBox1);

//приложение закрывается
Close();
```

Обработчик события **DropDownClosed** компонента **ComboBox**:

```
textBox3->Text=comboBox1->Items [comboBox1->
>SelectedIndex]->ToString();
```

8.3. Компонент WebBrowser

Компонент WebBrowser находится в окне **Панель элементов** в группе **Стандартные элементы управления** и предназначен для отображения **Web-страниц** в приложениях.

Если необходимо, чтобы после запуска приложения отображалась какая-либо страница сайта, то свойство **Url** компонента **WebBrowser** должно содержать адрес этого сайта. К примеру, записать в поле свойства **Url** значение <http://www.mail.ru>.

Для работы с компонентом используется множество методов. Рассмотрим некоторые из них:

- **GoBack()** – предыдущая страница в журнале переходов.
- **GoForward()** – следующая страница в журнале переходов.
- **GoHome()** – первая открытая страница.
- **Navigate()** – загрузка в **WebBrowser** страницы сайта, адрес которого указывается в скобках. Например,
`webBrowser1->Navigate ("http://www.rp5.by") ;`
- **Print()** – печать открытой Web-страницы.
- **Refresh()** – перезагрузка (обновление) открытой Web-страницы.
- **Stop()** – отмена незаконченных действий и динамических элементов на Web-странице.

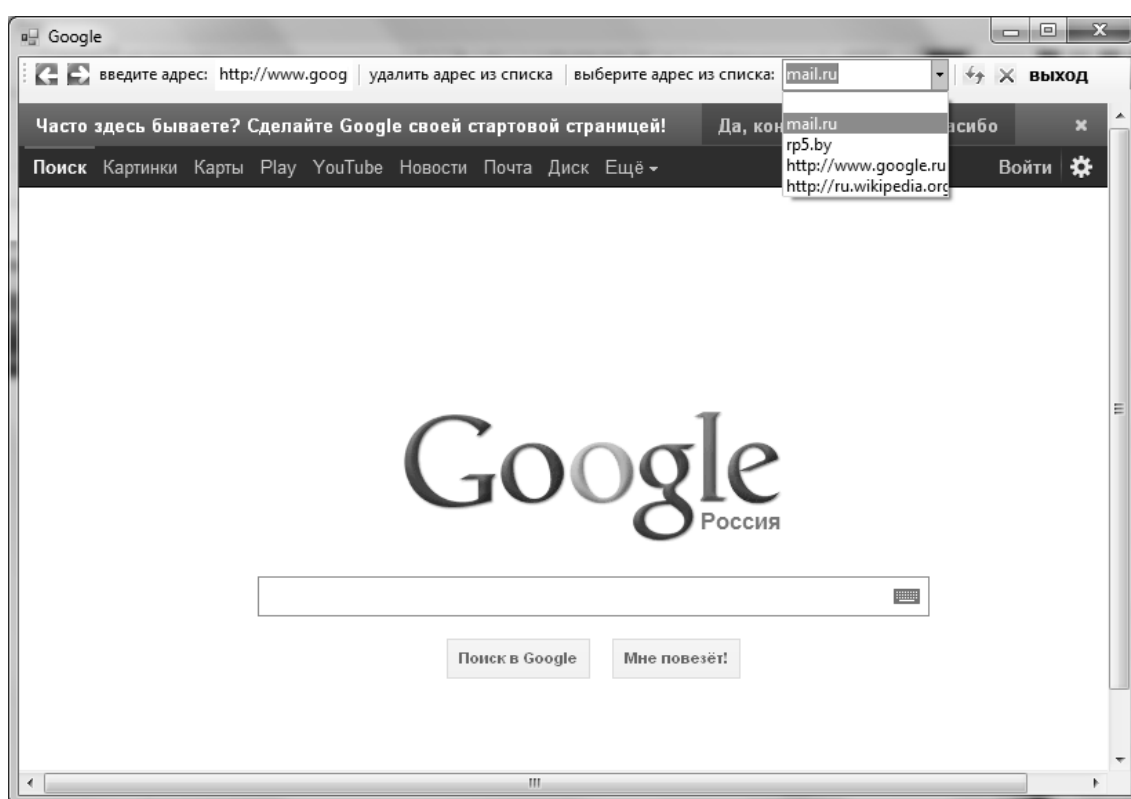
Задание для выполнения 3.

Разработать приложение для работы с сайтами. Рекомендуемый интерфейс приложения представлен на рисунке.

Требования к приложению:

- для работы с сайтами используются кнопки на панели инструментов;
- при запуске приложения список адресов открываемых файлов определяется содержимым файла, созданного заранее для их хранения;
- при открытии сайта его имя отображается в заголовке формы;

- файл открывается при выборе из списка адресов или при наборе его адреса в текстовом поле и нажатии клавиши **Enter**;
- перемещение между уже открывшимися страницами осуществляется с помощью соответствующих кнопок;
- список можно корректировать: добавлять новые файлы, вводя адрес в поле ввода, удалять – выбирая из списка и нажимая соответствующую кнопку **Удалить**.
- при закрытии формы измененный список адресов сохраняется в файле.



Рекомендации для выполнения задания.

- В папке разрабатываемого приложения, где находится файл **Form1.h** создайте текстовый файл **a.txt** с помощью программ **Блокнот** или **Word**. Запишите туда адреса сайтов, которые будут доступны для выбора в работающем приложении. Например, содержимое файла **a.txt** может выглядеть так:

[/mail.ru/rp5.by/http://www.google.ru/http://ru.wikipedia.org/](http://mail.ru/rp5.by/http://www.google.ru/http://ru.wikipedia.org/)

Если вы планируете использовать приложение для просмотра **xml**-файлов, размещенных на вашем компьютере, то файл можно наполнить, к примеру, следующим текстом:

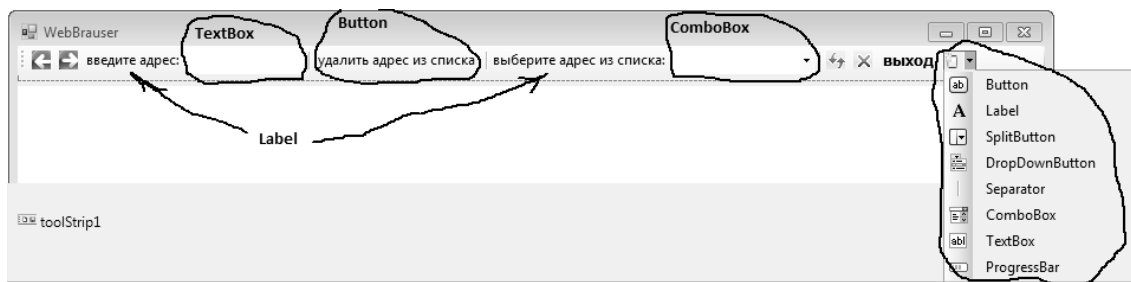
**D:\gr\Pshela.xml/D:\gr\Tarakan.xml/D:\gr\ Baboshka.xml/
D:\gr\Osa.xml/D:\gr\Muravey.xml/**

При этом обязательно используйте латинский шрифт для имен указываемых файлов и наклонную черту / для разделения имен в файлах.

Сохраните созданный файл.

– Добавьте на форму компоненты **splitContainer**, **toolStrip**, необходимые элементы на панель инструментов, **webBrowser**.

Чтобы наполнить компонент **toolStrip**, представляющий собой панель инструментов, необходимыми кнопками, надписями, текстовыми полями и выпадающими списками, нужно выбрать необходимый компонент из списка в правом углу панели, который появляется при выделении **toolStrip** в режиме **Конструктора**. При этом к имени всех добавленных на панель инструментов компонентов автоматически добавляется приставка **toolStrip**. Например, **toolStripComboBox1**, **toolStripTextBox1** и т.д.



– При загрузке приложения (событие формы **Load**) автоматически должен читаться созданный ранее текстовый файл **a.txt**, в котором находятся сведения о предыдущей работе приложения. Содержимым файла заполняется компонент **toolStripComboBox1**.

– При выборе строки в компоненте **toolStripComboBox1** она отображается в компоненте **toolStripTextBox1**. Содержимое

toolStripComboBox1 можно редактировать: добавлять новые строки, набранные в текстовом поле, удалять выбранные, используя соответствующие кнопки.

– Для работы с файлами использовать функции для работы с текстовыми файлами, разработанные при выполнении *задания 2*.

– Поместить функции в созданный заранее заголовочный файл или в файл **Form1.h** там же, где и при выполнении *задания 2*.

Данные функции будут отличаться тем, что вместо компонента **ComboBox** будет использоваться компонент **ToolStripComboBox**.

```
void RStrF(String ^F, ToolStripComboBox ^cb)
{
    String ^st, ^sb=File::ReadAllText(F);
    cb->Items->Clear();
    while(sb->Length>0)
    {
        int i=sb->IndexOf("/");
        if (i== -1) break;
        st=sb->Substring(0,i);
        cb->Items->Add(st);
        sb=sb->Substring(i+1, sb->Length-st->Length-1);
    }
    return; }

void SStrF(String ^F, ToolStripComboBox ^cb)
{String ^sa, ^sb;
int j=cb->Items->Count;
File::Delete(F);
for(int i=0;i<j;i++)
{
sa=cb->Items[i]->ToString();
sb+=sa->Concat(sa, "/");
}
File::AppendAllText(F, sb);
return;
}
```

Обработчик события **DropDownClosed** компонента

ToolStripComboBox:

//выполняется открытие выбранной страницы и отображение ее адреса

//в поле **toolStripTextBox1**

//если строка в toolStripComboBox1 не выделена

//на экране появится окно сообщения об этом

```
if (toolStripComboBox1->SelectedIndex==-1)
{
    MessageBox::Show("Выделите строку для удаления!",
    "Сообщение", MessageBoxButtons::OK,
    MessageBoxIcon::Error);
    return;
}
```

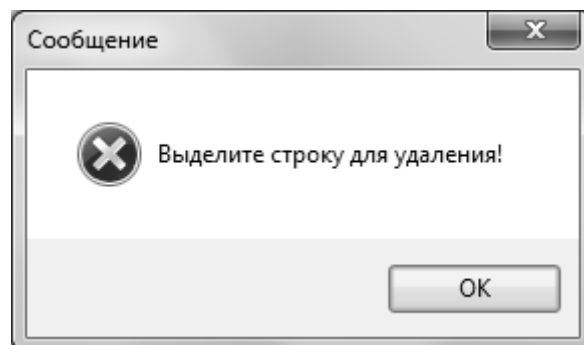
//выделенный элемент списка отобразится в текстовом поле адреса


```
toolStripTextBox1->Text=toolStripComboBox1->
Items[toolStripComboBox1->SelectedIndex]->
ToString();
```

//открытие страницы с указанным адресом

```
this->webBrowser1->Navigate(this->toolStripTextBox1->Text);
```

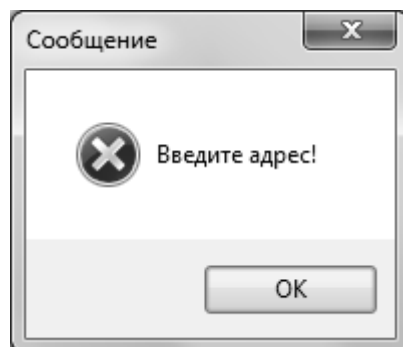
Вид окна сообщения при отсутствии выбора элемента в списке
ToolStripTextBox



Обработчик события «Нажатие на кнопку Обновить » панели инструментов **ToolStrip**:

```
//при обновлении страницы сначала проверяется – выбран ли адрес,  
//потом обновляется страница  
//если в выпадающем списке не выбран ни один элемент, то  
// появляется окно сообщения об этом  
    if (toolStripComboBox1->SelectedIndex==-1)  
    {  
        MessageBox::Show("Введите адрес!", "Сообщение",  
        MessageBoxButtons::OK, MessageBoxIcon::Error);  
    }  
    // перезагрузка (обновление) открытой Web-страницы  
    this->webBrowser1->Refresh();
```

Вид окна сообщения:



Обработчик события «Нажатие на кнопку **Удалить адрес их списка**» панели инструментов **ToolStrip**:

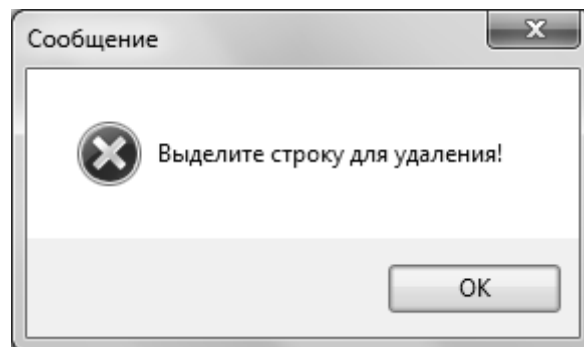
```
//если в выпадающем списке не выбран ни один элемент, то  
// появляется окно сообщения об этом
```

```
if (toolStripComboBox1->SelectedIndex==-1)  
{  
    MessageBox::Show("Выделите строку для удаления!",  
    "Сообщение", MessageBoxButtons::OK,  
    MessageBoxIcon::Error);  
}
```

```
//удаление выбранной строки из выпадающего списка
toolStripComboBox1->Items-
>Remove(toolStripComboBox1->SelectedItem);
```

```
//очистка текстового поля с указанным адресом
toolStripTextBox1->Clear();
```

Вид окна сообщения:




Обработчик события **KeyDown** (нажатие на клавишу) компонента **ToolStripComboBox**, используемого для ввода и отображения адреса открываемой **Web**-страницы:

//если при нахождении в фокусе текстового поля


//нажата клавиша **Enter**

```
    if (e->KeyCode==Keys::Enter)
    {
//строка текста из текстового поля добавляется в поле со списком
        toolStripComboBox1->Items->
        Add(toolStripTextBox1->Text);


// в WebBrowser загружается страница сайта, адрес которого
//указан в текстовом поле
        this->webBrowser1->Navigate(this->
        toolStripTextBox1->Text);
    }
```

Обработчик события «Нажатие на кнопку **Назад** » панели инструментов **ToolStrip**:

//загружается предыдущая страница в журнале переходов
`this->webBrowser1->GoBack()` ;

Обработчик события «Нажатие на кнопку **Вперед** » панели инструментов **ToolStrip**:

//загружается следующая страница в журнале переходов
`this->webBrowser1->GoForward()` ;

Обработчик события «Нажатие на кнопку **Остановить** » панели инструментов **ToolStrip**:

//отменяются незаконченные действия и динамические элементы на
//Web-странице
`this->webBrowser1->Stop()` ;

Обработчик события **DocumentCompleted** компонента **WebBrowser**, возникающего после окончательной загрузки страницы:

// в заголовке формы отображается имя открытой страницы
`Text=webBrowser1->DocumentTitle;`

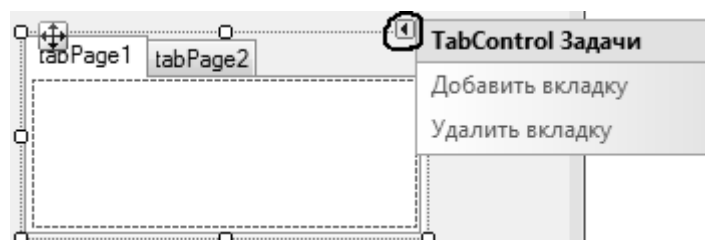
По желанию можно усложнить задание, добавив дополнительные функции в приложение, позволяющие, к примеру, открывать страницы на новых вкладках.

9.1. Компонент TabControl

Компонент **TabControl** расположен в окне **Панель элементов** в группе **Контейнеры** и используется для отображения информации на нескольких вкладках. Это позволяет не создавать несколько форм, необходимых для расположения отдельных групп информации, а использовать вместо этого вкладки на одной форме. С помощью компонента **TabControl** можно легко разработать многостраничное приложение.

После добавления на форму компонент **TabControl** содержит две вкладки. Добавить, отредактировать или удалить вкладки можно несколькими способами.

Так можно добавить или удалить вкладку, воспользовавшись списком задач, отображаемых при нажатии кнопки в левом верхнем углу компонента.

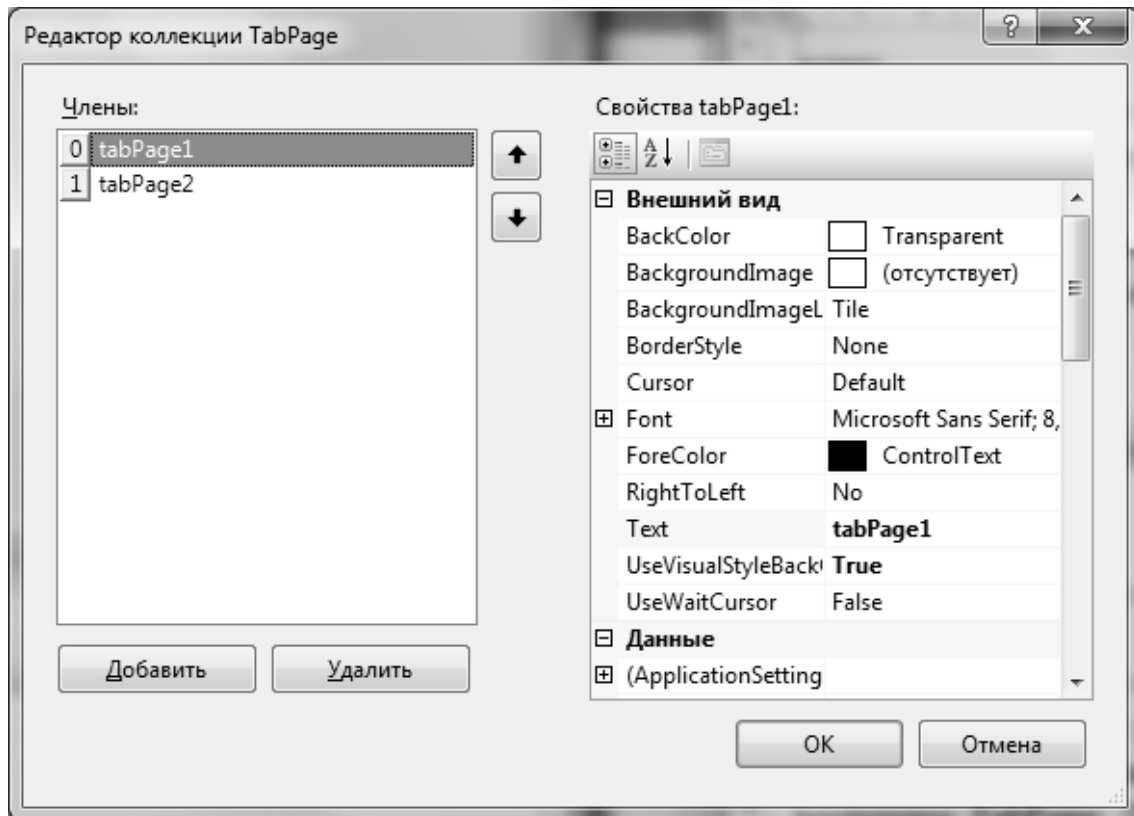


Кроме того можно выбрать данные операции в контекстном меню компонента.

Однако наиболее широкий доступ к возможностям по редактированию вкладок предоставляет свойство данного компонента **TabPagees**. При нажатии кнопки с многоточием в поле данного свойства открывается диалоговое окно **Редактора коллекции TabPage**, в котором можно не только удалять и добавлять вкладки, но и менять порядок их отображения, настраивать внешний вид и поведение каждой вкладки в отдельности и т.д.

Необходимо отметить, что доступ к свойствам и событиям отдельных вкладок можно получить в окне **Свойства**, предварительно

выделив вкладку, свойства которой нужно изменить. Выделить вкладку можно, открыв ее и выполнив щелчок левой кнопки мыши по ее центру.



Свойства всего компонента **TabControl** доступны, если выделена не отдельная вкладка, а весь компонент.

Некоторые свойства компонента **TabControl**:

- **Alignment** – расположение вкладок: **Top** – сверху, **Bottom** – снизу, **Left** – слева, **Right** – справа.



- **Appearance** – вид вкладок: **Normal** – стандартные вкладки, **FlatButtons** – плоские кнопки, **Buttons** – объемные кнопки. **FlatButtons** и **Buttons** можно использовать, если значение свойства **Alignment** установлено **Top**.

– **MultiLine** – размещение вкладок в несколько рядов.

У компонента **TabControl**, также как и у остальных компонентов, многие свойства недоступны из окна **Свойства** и устанавливаются программно:

– **SelectedIndex** – номер текущей страницы. Следует учитывать, что первая страница вкладок имеет **SelectedIndex** 0. При отсутствии выбранной страницы **SelectedIndex** равно -1.

Таким образом, чтобы открыть первую страницу достаточно записать:

```
tabControl1->SelectedIndex=0;
```

– **SelectedTab** – текущая страница компонента **TabControl**.

Используя это свойство можно выполнить то же действие:

```
tabControl1->SelectedTab=tabPage1;
```

9.2. Компоненты **DateTimePicker**, **Timer**, **ProgressBar**

Компонент **DateTimePicker**

Компонент **DateTimePicker** расположен в окне **Панель элементов** в группе **Стандартные элементы управления** и применяется для выбора и отображения даты или времени в нужном формате.

Некоторые свойства компонента **DateTimePicker**:

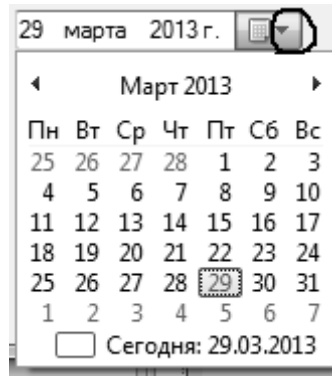
– **Format** – вид отображаемых параметров: **Long** – дата в длинном формате, **Short** – дата в коротком формате, **Time** – время, **Custom** – пользовательский формат, устанавливаемый в свойстве **CustomFormat**.

– **ShowUpDown** – вариант изменения даты/времени: **true** – с помощью кнопки **up-down** выделенная часть изменяется на одно значение;



false – для выбора даты используется выпадающий список в виде ка-

лендаря



- **Value** – значение выбранной даты или времени.
- **MaxDate** и **MinDate** – максимальная и минимальная возможные даты.

Компонент **Timer**

Компонент **Timer** расположен в окне **Панель элементов** в группе **Компоненты** и применяется в качестве счетчика времени, по истечении которого происходит некоторое событие. В работающем приложении данный компонент на форме не отображается.

Некоторые свойства компонента **Timer**:

- **Enabled** – запуск (**true**) и остановка (**false**) таймера.
- **Interval** – время, по истечении которого выполняется событие **Tick** и повторяются действия с ним связанные.

Запустить и остановить таймер можно не только с помощью свойства **Enabled**, но и с помощью методов **Start ()** – запуск и **Stop ()** – остановка таймера.

Компонент **ProgressBar**

Компонент **ProgressBar** расположен в окне **Панель элементов** в группе **Стандартные элементы управления** и применяется для индикации различных процессов.

Некоторые свойства компонента **ProgressBar**:

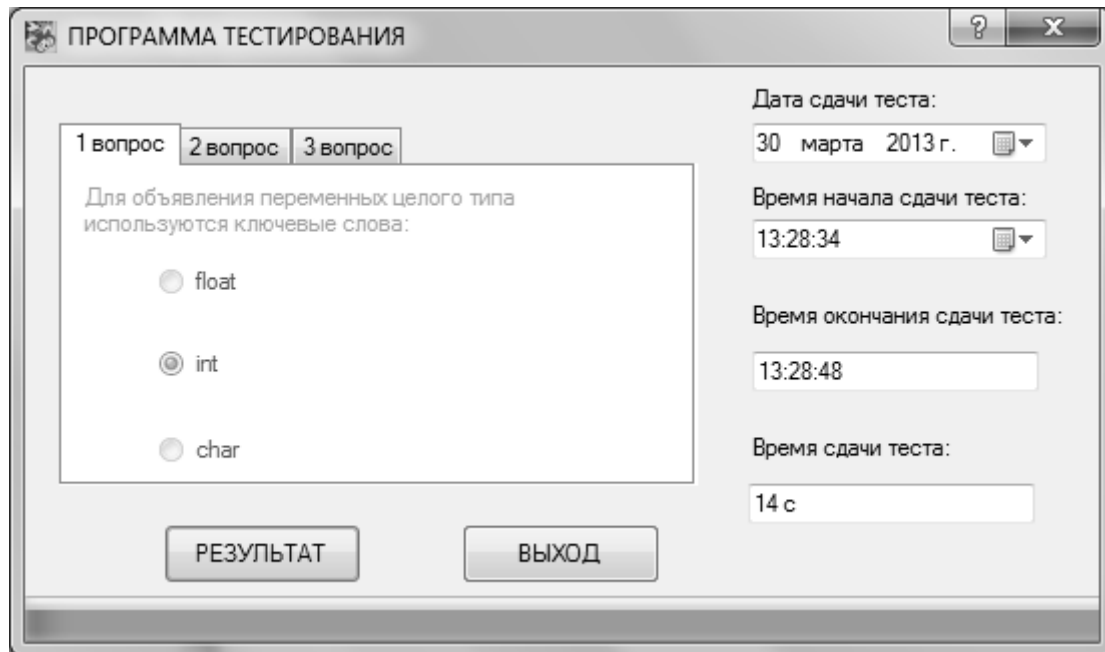
- **Minimum** – минимальное значение.
- **Maximum** – максимальное значение.
- **Value** – текущее значение.
- **Step** – интервал, с которым увеличивается **Value**.

Для обеспечения функционирования компонента **ProgressBar** используется компонент **Timer**, в обработчике события **Tick** которого увеличивается значение свойства **Value**.

Задание для выполнения.

Разработать программу тестирования.

Рекомендуемый интерфейс приложения приведен на рисунке.



Рекомендации по разработке приложения:

При запуске приложения появляется форма заставка, которая раскрывается на весь экран и закрывается через 5 секунд, постепенно теряя прозрачность.

Для реализации необходимо:

- добавить в приложение новую форму **Form2**;
- подключить данную форму к первой, добавив в файл **Form1.h**
`#include "Form2.h"`

– добавить на форму два компонента **Timer** и один компонент **PictureBox**;

- для **timer1** установить свойства:

Enabled – true;

Interval – 5000;

– для **timer2** установить свойства:

Enabled – true;

Interval – 500;

– для **Form2** установить свойства:

WindowState – Maximized;

StartPosition – CenterScreen;

FormBorderStyle – None;

– для **pictureBox1** установить свойства:

Image – добавить любую картинку;

SizeMode – StretchImage.

– Чтобы при загрузке формы размер картинки устанавливался равным размеру формы, необходимо использовать обработчик события загрузки формы (**Load**)

```
pictureBox1->Width=Width;
```

```
pictureBox1->Height=Height;
```

– Чтобы форма постепенно теряла прозрачность, нужно использовать **обработчик события timer2**, которое происходит по истечении заданного временного интервала (**Tick**). **Opacity** – задает степень прозрачности формы (**1** – непрозрачная, **0** – полностью прозрачная).

```
this->Opacity=Opacity-0.1;
```

– Чтобы форма закрылась по истечении 5 секунд, нужно использовать обработчик события **timer1**, которое происходит по истечении заданного временного интервала (**Tick**).

```
Close();
```

– Вторая форма открывается при первой загрузке первой формы. Для реализации необходимо использовать обработчик события **Shown** формы **Form1**:

```
Form2^ f2=gcnew Form2();  
f2->Show();
```

Для организации процесса тестирования:

- на первую форму добавить компоненты: **tabControl1**, кнопки «РЕЗУЛЬТАТ» и «ВЫХОД».
- добавить вкладки к компоненту **tabControl1**, чтобы их общее количество было равно **3**.
- подписать вкладки, кликая внутри каждой и изменяя свойство **Text** (1 вопрос, 2 вопрос, 3 вопрос).
- поместить на каждую из вкладок вопросы теста, используя компоненты **radioButton** и установив значение их свойства **Checked** равным **false**:

Для объявления переменных целого типа
используются ключевые слова:

☐ float

☐ int

☐ char

Правильный ответ 2.

Цикл с постусловием реализуется с помощью оператора:

☐ for

☐ while

☐ do while

Правильный ответ 3.

```
int y=5%2;
```

Результат выполнения равен:

☒ 1

☐ 2

Правильный ответ 1.

– получить результат тестирования по нажатию на кнопку «РЕЗУЛЬТАТ» можно в результате использования следующего обработчика события (номера компонентов **radioButton** устанавливать в соответствии с правильными ответами).

Используется вспомогательная переменная **t**, к которой прибавляется **1** в случае выбора правильного ответа. В зависимости от конечного значения **t** определяется значение **rez**, которое выводится в окне сообщения **MessageBox**:

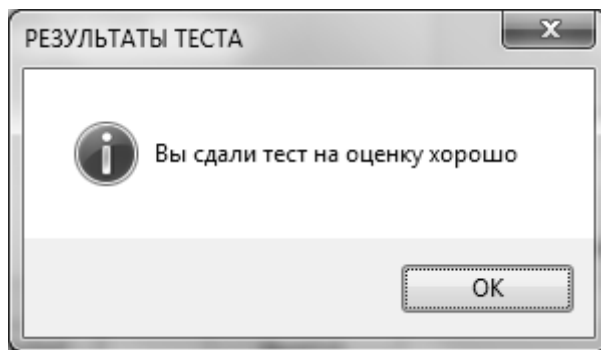
Обработчик события «Нажатие на кнопку **Результат**»

```
int t=0;
if (radioButton2->Checked==true) t++;
if (radioButton4->Checked==true) t++;
if (radioButton9->Checked==true) t++;
String^ rez;

switch(t)
{
    case 1:rez="удовлетворительно";break;
    case 2:rez="хорошо";break;
    case 3:rez="отлично";break;
    default:rez="неудовлетворительно";
}

MessageBox::Show("Вы сдали тест на оценку
"+rez,"РЕЗУЛЬТАТЫ ТЕСТА", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
```


Результат выполнения обработчика события:



Для организации переключения вкладок по истечении 3 секунд:

– на форму **Form1** добавить **timer1**.

Установить свойства:

Enabled – false,

Interval – 3000.

– Объявить переменную **i** и присвоить ей значение **0** после строки

```
using namespace System::Drawing;  
int i=0;
```

– Использовать обработчик события **timer1**, которое происходит по истечении заданного временного интервала (**Tick**).

//если **i** меньше общего числа вкладок в **tabControl1**

//по истечении заданного интервала

```
if (i<tabControl1->TabCount)
```

```
{
```

//недоступна текущая вкладка

```
tabControl1->SelectedTab->Enabled=false;
```

//переход на следующую вкладку

```
tabControl1->DeselectTab(i);
```

```
i++; }
```

```
else
```

//отключение первого таймера

```
timer1->Enabled=false;
```

Для синхронного запуска процесса переключения вкладок и индикации тестирования (заполнение прямоугольниками компонента **progressBar**):

- На форму добавить **timer2** и **timer3**.

- Установить свойства **timer2**:

Enabled – true,

Interval – 5000.

- Использовать обработчик события **timer2** , которое происходит по истечении заданного временного интервала (**Tick**) и включает таймеры **1** и **3**, отключает таймер **2**.

```
timer2->Stop();  
timer1->Enabled=true;  
timer3->Enabled=true;
```

Чтобы организовать работу компонента **progressBar1** по отображению хода процесса:

- Добавить на форму компонент **progressBar1**.

- Установить свойства **timer3**:

Enabled – false,

Interval – 300.

- Установить свойства **progressBar1**:

Maximum – 30.

- Использовать обработчик события **timer3** , которое происходит по истечении заданного временного интервала (**Tick**)

```
//Изменить текущее значение progressBar1  
progressBar1->Value=progressBar1->Value+1;
```

```
//остановка таймера по достижению Maximum = 30  
if (progressBar1->Value==progressBar1->Maximum)  
    {timer3->Stop();}
```

Для отображения информации о дате и времени использовать компоненты **dateTimePicker** и функции для работы с датами и временем

The screenshot shows a form with four controls. The first control is a 'DateTimePicker' labeled 'Дата сдачи теста:' showing '8 декабря 2010 г.'. The second is a 'DateTimePicker' labeled 'Время начала сдачи теста:' showing '20:53:47'. The third is a 'TextBox' labeled 'Время окончания сдачи теста:'. The fourth is a 'TextBox' labeled 'Время сдачи теста:'. Callouts point to each control with their respective names and formats: 'dateTimePicker1 Format - Long' for the first, 'dateTimePicker2 Format - Time' for the second, 'textBox1' for the third, and 'textBox2' for the fourth.

– Добавить на форму два компонента **dateTimePicker**, изменить свойство **Format**.

– Изменить обработчик события **timer3**, которое происходит по истечении заданного временного интервала (**Tick**), добавив недостающие строки:

Обработчик события **Tick** компонента **timer3**, расположенного на первой форме:

```
progressBar1->Value=progressBar1->Value+1;
```

```
if (progressBar1->Value==progressBar1->Maximum)
{
timer3->Stop();
}
```

```
// d1= текущее время
```

```
System::DateTime d1=System::DateTime::Now;
```

```
//d2= время в компоненте dateTimePicker2
```

```
//устанавливается при открытии формы
```

```
System::DateTime d2=dateTimePicker2->Value;
```

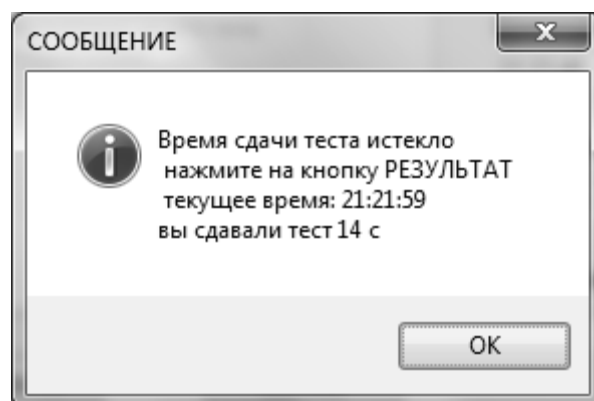
```
//вывод d1 в поле в форматированном виде
textBox1->Text=d1.ToString("HH:mm:ss");

//определение разности
// System::TimeSpan – класс, представляющий интервал времени
System::TimeSpan dr=d1-d2;
textBox2->Text=dr.Seconds.ToString()+" с";
// dr.Seconds – количество полных секунд
String ^s1;

//s1 равно d1 в форматированном виде
s1=d1.ToString("HH:mm:ss");

//вывод сообщения
MessageBox::Show("Время сдачи теста истекло \n
нажмите на кнопку РЕЗУЛЬТАТ\n
текущее время:
"+s1+"\n
вы сдавали тест "+dr.Seconds+" с"
,"СООБЩЕНИЕ",MessageBoxButtons::OK,MessageBoxIcon::
:Asterisk);
```

После выполнения данного обработчика событий появится диалоговое окно сообщения:



В текстовые поля выведется время окончания теста и время сдачи теста.

Глава 10.

Обработка табличной информации

10.1. Массивы CLR

В отличие от массивов C++ массивы CLR располагаются в памяти с автоматической сборкой мусора и имеют набор методов для их обработки.

Объявляются данные массивы с помощью ключевого слова **array**. Причем тип данных элементов указывается в угловых скобках, а переменные массивов – всегда отслеживаемые дескрипторы.

Например, объявить массив **CA** вещественных чисел можно так:
`array<float>^ CA;`

или создать данный массив для хранения 50 элементов при объявлении. Количество элементов указывается в круглых скобках:

```
array<float>^ CA= gcnew array<float>(50);
```

Первый элемент имеет индекс **0**, поэтому для заполнения массива значениями, рассчитанными по формуле **5·km+1**, где **km** – индекс массива, достаточно записать:

```
for(int km=0; km<50; km++)  
    CA[km]=5*km+1;
```

То же действие можно выполнить, используя такое свойство массива, как его длина – **Length**, определяющее количество элементов массива.

```
for(int km=0; km<CA->Length; km++)  
    CA[km]=5*km+1;
```

Массив можно создать, инициализировав его при объявлении:
`array<int>^ MA={378,125,789,451,545,974,244};`

В данном случае размер массива равен 7, согласно количеству элементов, указанных в фигурных скобках.

Можно использовать любой тип элементов массива. Например, создать массив строк из четырех элементов:

```
array<String^>^ ST={"Spring", "Summer", "Autumn",  
"Winter"};
```

или

```
array<String^>^ ST;  
ST=gcnew array<String^>{"Spring", "Summer",  
"Autumn", "Winter"};
```

Базовым для всех массивов CLR является класс **Array**, которому принадлежит множество методов для работы с массивами. Рассмотрим некоторые из них:

– **Clear()** – обнуляет заданные элементы массива. Следующая запись обнуляет все элементы массива **CA**

```
Array::Clear(CA, 0, CA->Length);
```

CA – имя массива; **0** – индекс первого обнуляемого элемента; **CA->Length** – размер массива **CA**.

– **Sort()** – сортировка массива по возрастанию.

```
Array::Sort(CA);
```

Существует несколько версий данной функции: с указанием количества сортируемых элементов, одновременной сортировки двух массивов и т.д.

– **BinarySearch()** – поиск в одномерном массиве, отсортированном заранее.

Многомерные массивы CLR

Чтобы создать двумерный массив целых чисел **CA** размерностью четыре строки и семь столбцов, нужно записать:

```
array<int, 2>^ CA=gcnew array<int, 2>(4, 7);
```

Из записи видно, что размерность массива указывается в угловых скобках через запятую после указания типа данных элементов

массива, а количество строк и столбцов в круглых скобках при создании данного массива.

Рассчитать значения элементов массива по формуле $(4 \cdot m_i + 1) \cdot (2 \cdot m_j + 1)$, где m_i – индекс строки, а m_j – индекс столбца, можно следующим образом:

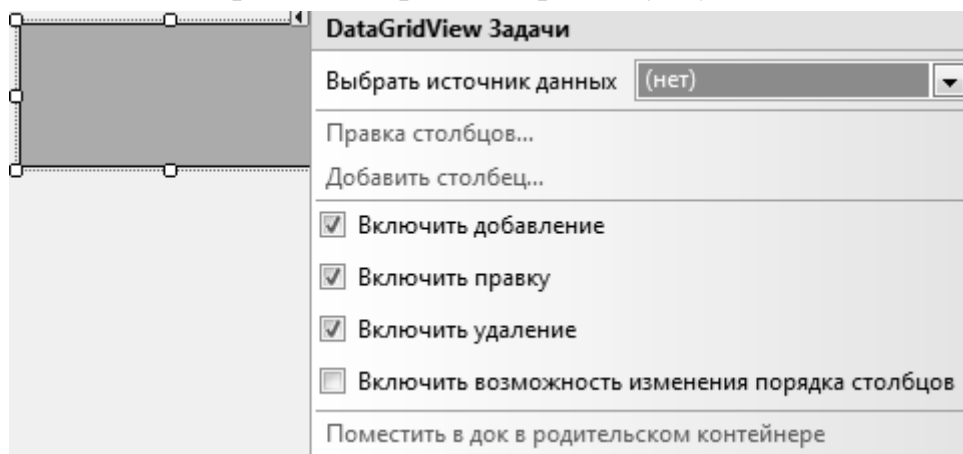
```
int nr=4; // количество строк
int nc=7; // количество столбцов
array<int,2>^ CA=gcnw array<int,2>(nr,nc);

for (int mi=0;mi<nr;mi++)
for (int mj=0;mj<nc;mj++)
CA[mi,mj]=(4*mi+1)*(2*mj+1);
```

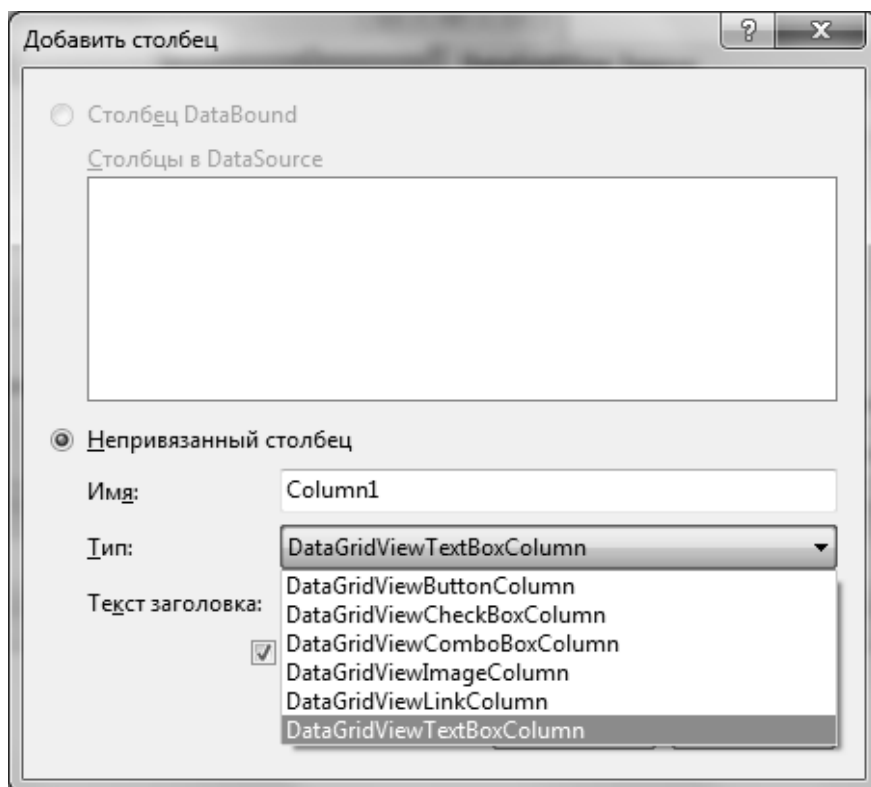
10.2. Работа с компонентом DataGridView

Для отображения элементов двумерных массивов наиболее целесообразно использовать компонент **DataGridView**, расположенный в окне **Панель элементов** в группе **Данные**. В данном компоненте можно отображать любые данные, включая изображения, располагая их в массиве ячеек, находящихся на пересечении строк и столбцов, которые имеют заголовки. Элемент управления **DataGridView** может быть связан с источником данных, здесь данный случай не рассматривается.

После добавления компонента на форму у него отсутствуют строки и столбцы. Создать их можно, вызвав список задач, путем нажатия кнопки со стрелкой в правом верхнем углу компонента.



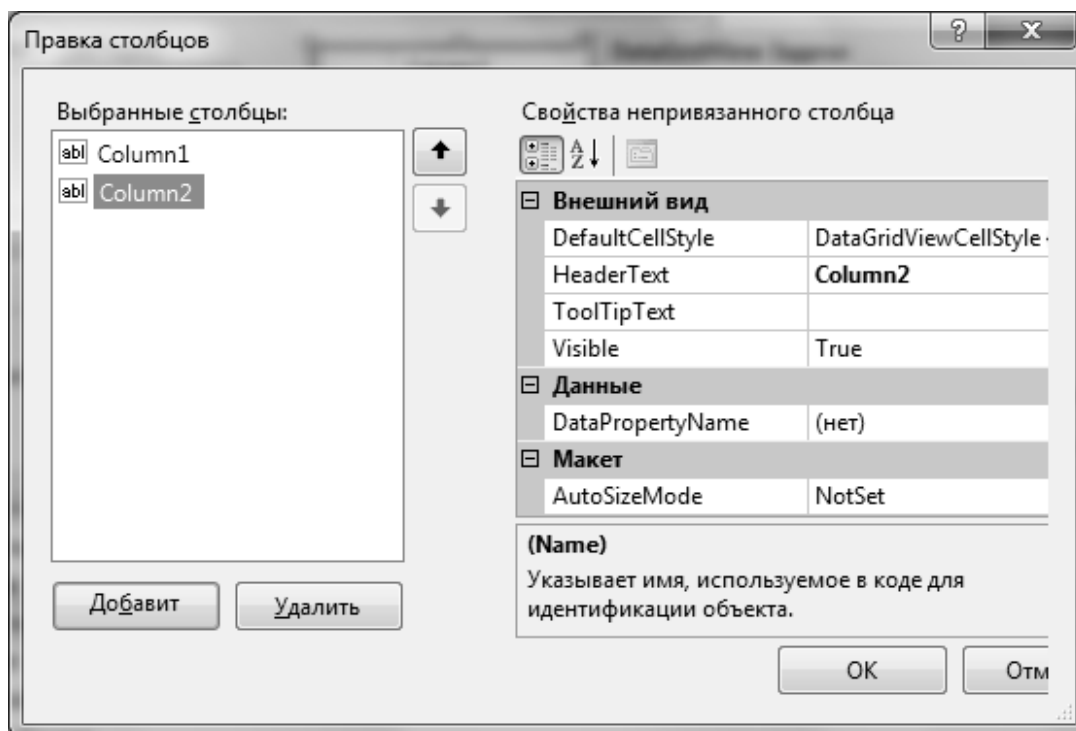
Добавить столбец можно выбрав пункт **Добавить столбец...**, в котором можно определить имя и тип данных столбца, а также текст его заголовка.



Типы столбцов компонента **DataGridView** следует выбирать в зависимости от того, в каком виде будут представлены данные, которые хранятся в ячейках:

- **DataGridButtonColumn** – кнопки.
- **DataGridCheckBoxColumn** – флажки.
- **DataGridComboBoxColumn** – выпадающий список.
- **DataGridImageColumn** – изображения.
- **DataGridLinkColumn** – ссылки.
- **DataGridTextBoxColumn** – текстовое поле.

Изменить свойства столбцов, добавить и удалить их можно с помощью диалогового окна **Правка столбцов...**, которое открывается при выборе соответствующего пункта в списке задач компонента **DataGridView**. В данном окне можно изменить параметры каждого столбца в отдельности, выбрав его имя в левом поле окна: **HeaderText** – текст заголовка, **ToolTipText** – текст подсказок, **Visible** – видимость столбца, **Width** – ширина, **SortMode** – способ сортировки и другие, рассматриваемые ниже.



Многие свойства компонента **DataGridView** устанавливаются как в окне **Свойства**. К примеру, после нажатия в поле свойства **Columns** кнопки с многоточием, открывается диалоговое окно **Правка столбцов...**

Однако некоторые свойства можно изменить только программно. Все методы и свойства, которые можно применить к данному компоненту отображаются с помощью суфлера кода после записи в обработчике события

```
DataGridView->
```

Далее можно выбрать нужное с помощью щелчка левой кнопки мыши или нажатия клавиши **Enter**. При наведении курсора на выбранное свойство или метод появится всплывающая подсказка, поясняющая его назначение.

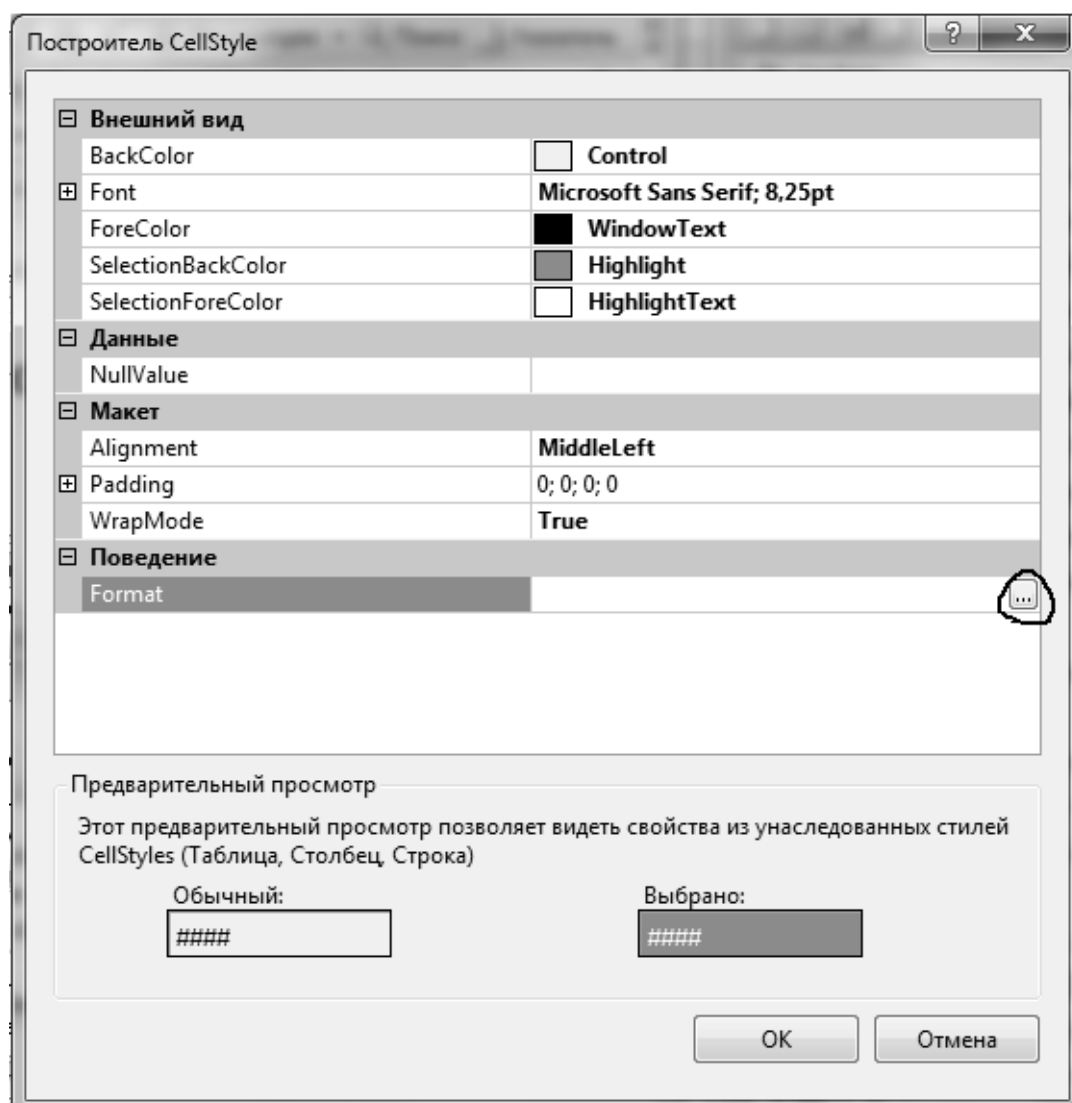
Рассмотрим некоторые свойства компонента **DataGridView**.

- **CellStyle** – вид границы компонента: **Custom** – настраиваемая, **Single** – одна линия, **Raised** – трехмерная выпуклая, **Sunken** – трехмерная утопленная, **None** – без границ, **SingleVertical** – вертикальная однолинейная, **RaisedVertical** – вертикальная трехмерная выпуклая, **SunkenVertical** – вертикальная трехмерная утоп-

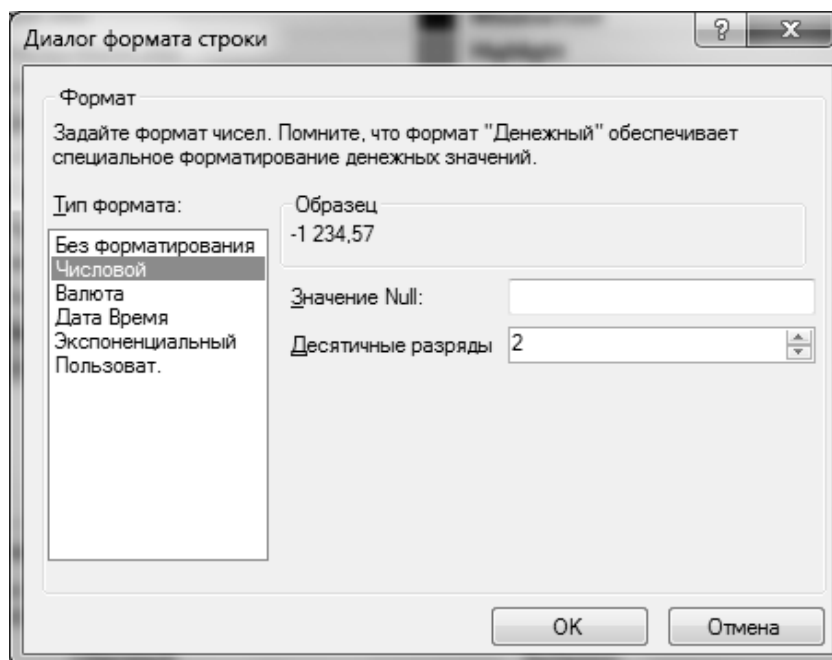
ленная, **SingleHorizontal** – горизонтальная однолинейная, **RaisedHorizontal** – горизонтальная трехмерная выпуклая, **SunkenHorizontal** – горизонтальная трехмерная утопленная.

– **ColumnHeadersBorderStyle** – вид границы заголовков столбцов.

– **ColumnHeadersDefaultCellStyle** – оформление заголовка столбца, применяемое по умолчанию. При нажатии кнопки с многоточием в поле данного свойства открывается диалоговое окно **Построитель CellStyle**, в котором можно настраивать внешний вид как ячеек заголовка столбца, устанавливая цвет фона(**BackColor**), вид (**Font**) и цвет(**ForeColor**) шрифта, их изменение при выделении ячеек (**SelectionBackColor**, **SelectionForeColor**), выравнивание текста(**Alignment**), возможность переноса по словам(**WrapMode**) и формат отображаемых данных(**Format**).



Настроить формат данных можно в диалоговом окне, открываемом при нажатии кнопки с многоточием в поле данного свойства.



- **ColumnHeadersVisible** – отображение заголовков столбцов.
- **DefaultCellStyle** – стиль оформления ячейки по умолчанию. Используется, если стиль не определен с помощью других свойств. Устанавливается с помощью диалогового окна **Построитель CellStyle**, рассмотренного ранее. Нужно внимательно подходить к установке свойств по умолчанию для ячейки, строки и столбца, чтобы получить требуемый вариант оформления, так как ячейка всегда находится на пересечении строки и столбца.
- **GridColor** – цвет линий ячеек компонента.
- **MultiSelect** – возможность одновременного выделения нескольких ячеек.
- **ReadOnly** – запрет на редактирование ячеек.
- **RowHeadersBorderStyle** – вид границы заголовков строк.
- **RowHeadersDefaultCellStyle** – оформление заголовков строк, выполняется так же как и для столбцов при помощи диалогового окна **Построитель CellStyle**, в котором можно настраивать внешний вид ячеек.
- **RowHeadersVisible** – отображение заголовков строк.

– **RowHeadersDefaultCellStyle** – оформление заголовка строк, применяемое по умолчанию, используется диалоговое окно **Построитель CellStyle**.

– **SelectedCells**, **SelectedColumns**, **SelectedRows** – выделенные ячейки, столбцы, строки.

– **SelectionMode** – способ выделения ячеек: **CellSelect** – одна или несколько ячеек щелчком левой кнопки мыши по ним, **FullRowSelect** – вся строка щелчком по заголовку или по любой ячейке данной строки, **RowHeaderSelect** – строка щелчком по заголовку, **FullColumnSelect** – весь столбец щелчком по заголовку или по любой ячейке данного столбца, **ColumnHeaderSelect** – весь столбец щелчком по заголовку.

– **Columns** – все столбцы компонента. Позволяет обращаться к отдельным столбцам, изменяя их свойства. К примеру, изменить подписи заголовков столбцов:

```
dataGridView1->Columns[0]->HeaderText="первый";  
dataGridView1->Columns[1]->HeaderText="второй";  
dataGridView1->Columns[2]->HeaderText="третий";
```

– **Rows** – все строки компонента.

– **Cells[i]** – *i*-ая ячейка строки. Нумерация строк, столбцов и ячеек начинается с нуля, поэтому чтобы обратиться к первой ячейке второй строки нужно записать:

```
dataGridView1->Rows[1]->Cells[0]->Value=5;
```

В результате в данную ячейку будет занесено значение 5.

Так же можно присвоить и строковые данные:

```
dataGridView1->Rows[1]->Cells[0]->Value="ответ";
```

– **ColumnCount** – количество столбцов.

– **RowCount** – количество строк.

Чтобы программно установить количество строк равным 5 и количество столбцов равным 4 достаточно записать:

```
dataGridView1->ColumnCount=4;  
dataGridView1->RowCount=5;
```

Некоторые методы компонента **DataGridView** для работы со строками:

– **Add()**, **Insert ()** – добавить строку. Необходимо учитывать количество столбцов компонента. Например, добавить строку в компонент **DataGridView** с четырьмя столбцами можно так:

```
dataGridView1->Rows->Add("Spring", "Summer",  
"Autumn", "Winter");
```

– **Clear()** – очистить все. Пример, как удалить все строки.

```
dataGridView1->Rows->Clear();
```

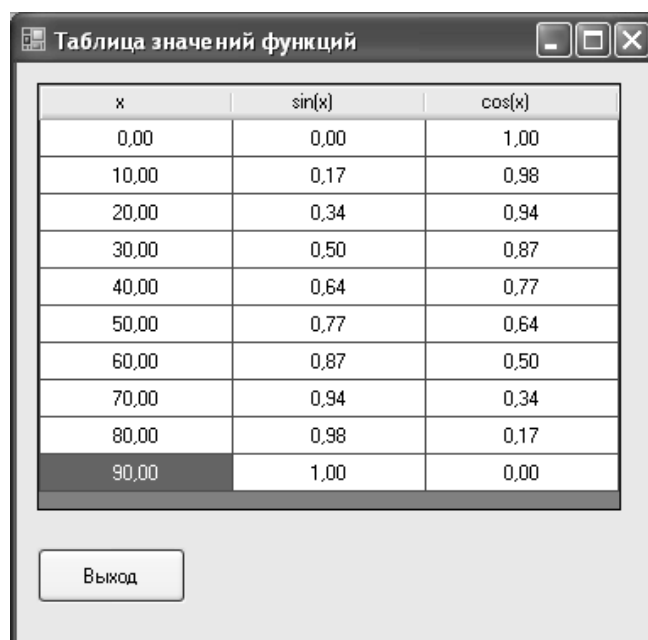
– **AddCopy()** – добавить копию.

– **InsertCopy()** – вставить копию.

– **Remove ()** – удалить строку.

Задание для выполнения.

Разработать приложение, определяющее таблицу значений функций **sin(dx)** и **cos(dx)** для **dx**, изменяющегося от **0** до **90** градусов с шагом **10**. Рекомендуемый интерфейс приложения представлен на рисунке.



Рекомендации для выполнения задания:

- таблица значений должна заполняться при загрузке формы;
- добавить столбцы к компоненту **dataGridView1** с помощью

команды его контекстного меню **Правка столбцов...**;

– в открывшемся окне необходимо:

изменить текст в ячейке заголовков добавленных столбцов(**HeaderText**);

тип столбца(**ColumnType**) установить – **DataGridViewTextBoxColumn**;

стиль ячейки столбца по умолчанию(**DefaultCellStyle...** – нажать многоточие, откроется окно **Построитель CellStyle** – по желанию изменить внешний вид, выравнивание в столбце – **Alignment** – **MiddleCenter**, формат строки для 2 и 3 столбца установить с помощью **Format**, нажав на многоточие и выбрав **Числовой**, десятичных разрядов - 2);

– установить свойства строк:

свойство компонента **dataGridView1** – **RowDefaultCellStyle...**

– спрятать столбец, содержащий заголовки строк

`RowHeadersVisible=false;`

Обработчик события формы **Shown()**:

```
//устанавливает количество строк=10
```

```
dataGridView1->RowCount=10;
```

```
int dx=0;
```

```
for(int di=0;di<10;di++)
```

```
{//заполняет первый столбец - индекс 0
```

```
dataGridView1->Rows[di]->Cells[0]->Value=dx;
```

```
//второй столбец
```

```
dataGridView1->Rows[di]->Cells[1]->Value=
```

```
Math::Sin(dx*Math::PI/180);
```

```
//третий столбец
```

```
dataGridView1->Rows[di]->Cells[2]->Value=
```

```
Math::Cos(dx*Math::PI/180);
```

```
//изменение шага
```

```
dx=dx+10; }
```

Глава 11

Добавление компонентов на форму в процессе выполнения приложения

11.1. Добавление компонентов и установка их свойств в режиме кода

До сих пор мы добавляли компоненты на форму в режиме **Конструктора**, используя **Панель элементов**. Однако это не всегда удобно. К примеру, на форме в разные моменты времени и при выполнении различных событий должны отображаться различные изображения, надписи и т.д. Причем это должно происходить на ограниченном пространстве. Выходом из данной ситуации может стать добавление компонентов и изменение их свойств программным образом.

Задание для выполнения 1.

Разработать приложение, в котором при нажатии на кнопку динамически создается надпись «**ДИНАМИЧЕСКАЯ НАДПИСЬ**» белого цвета на красном фоне. Кнопка добавлена на форму в режиме **Конструктора**, надпись – программно.

Для выполнения задания на форму **Form1** добавить кнопку и внести изменения в файл **Form1.h**, добавив текст, выделенный жирным шрифтом и серым фоном, **строго в указанные места кода**. Ориентиром может служить ссылка на компонент **Button**, созданная автоматически.

Form1.h

```
#pragma once
namespace DinLab{
    . . .

    //эта строка имеется в файле Form1.h
    private: System::Windows::Forms::Button^ button1;

    //данную строку нужно добавить самостоятельно
    /*объявление создаваемой надписи*/
    private: System::Windows::Forms::Label^ dinlab;
```

```

. . .
//эта строка имеется в файле Form1.h
this->button1=(gcnew System::Windows::Forms::
Button()) ;
//данную строку нужно добавить самостоятельно
//создать надпись, используя операцию gcnew
this->dinlab=(gcnew
System::Windows::Forms::Label()) ;

. . .
#pragma endregion
//Обработчик событие нажатие на кнопку на форме
//выполните двойной щелчок по кнопке в режиме Конструктора и
//впишите текст туда, где находится курсор

//определение параметров созданной надписи
//изменение размера компонента в соответствии с размером
//используемого шрифта
this->dinlab->AutoSize = true;

//координаты левого верхнего угла компонента

this->dinlab->Location
=System::Drawing::Point(73,72) ;

//имя создаваемого компонента
this->dinlab->Name = L"dinlab";

//размер данного элемента управления в точках
this->dinlab->Size =System::Drawing::Size(41, 22) ;

//фоновый цвет компонента
this->dinlab->BackColor=Color::Red;

//цвет шрифта компонента

```



```
this->dinlab->ForeColor=Color::White;
```

//текст надписи

```
this->dinlab->Text = L"ДИНАМИЧЕСКАЯ НАДПИСЬ";
```

// на форму добавляется надпись

```
this->Controls->Add(this->dinlab);
```

//можно добавить различные свойства надписи по своему вкусу

11.2. Обработка событий компонентов, добавленных в процессе выполнения программы

Задание для выполнения 2.

Усовершенствовать приложение, разработанное в результате выполнения *задания 1*. При щелчке по созданной надписи должен меняться цвет надписи с красного на синий.

Рекомендации для выполнения задания:

– Добавить приведенный ниже текст в обработчик события «нажатие на кнопку **Button1**», созданный в *задании 1*:

// означает, что у надписи имеется событие Click:

```
this->dinlab->Click+=gcnew      System::EventHandler  
(this, &Form1::dinlab_Click);
```

– Определить событие «Щелчок по надписи» после вышеописанного обработчика события «нажатие на кнопку **Button1**»
(ЗАПИСАТЬ ВЕСЬ ТЕКСТ С ЗАГОЛОВКОМ)

```
private: System::Void dinlab_Click(System::Object^  
sender, System::EventArgs^ e)  
{ this->dinlab->ForeColor=Color::Blue; }
```

Задание для выполнения 3.

Разработать приложение, при загрузке которого в форме появляется изображение и кнопка.

Рекомендации по выполнению задания.

– Использовать динамическое создание объектов. Файл

изображения должен быть сохранен на диске **d** под именем **777.jpg**.

- **Не добавлять компоненты в режиме Конструктора.**
- При щелчке по изображению появляется динамическая надпись «**ЭТО ДИНАМИЧЕСКАЯ КАРТИНКА**».
- При щелчке по кнопке изображение прячется.
- При щелчке по форме изображение появляется.
- Для выполнения задания необходимо добавить в файл **Form1.h** выделенный текст.

```
#pragma once
namespace din {
    . . .
public ref class Form1:public
System::Windows::Forms::Form
{
    public:
        Form1(void)
        {
            InitializeComponent();
        }
    protected:
        ~Form1()
        {if (components)
            {delete components;}}

/*объявление динамически создаваемых объектов*/
private: System::Windows::Forms::Label^  dinlab;
private: System::Windows::Forms::Button^  but;
private: System::Windows::Forms::PictureBox^
dinpib;
    . . .
        void InitializeComponent(void)
        {
/*создание динамических объектов*/
this->dinpib=(gcnew
System::Windows::Forms::PictureBox());
```

```
this->but=(gcnew  
System::Windows::Forms::Button());
```

```
this->dinlab=(gcnew  
System::Windows::Forms::Label());
```

```
. . .
```

```
#pragma endregion
```

//Обработчик события Shown формы Form1

//задаются параметры dinpib

```
{  
this->dinpib->  
>Location=System::Drawing::Point(121,12);  
this->dinpib->Name = L"dinpib";  
this->dinpib->Size = System::Drawing::Size(298,  
240);  
this->dinpib->SizeMode=System::Windows::Forms::  
PictureBoxSizeMode::StretchImage;  
this->dinpib->  
>Image=Image::FromFile("d:\\777.jpg");  
this->Controls->Add(this->dinpib);
```

//создается событие "щелчок кнопкой мыши"

```
this->dinpib->MouseDown += gcnew  
System::Windows::Forms::EventHandler(this,  
&Form1::dinpib_MouseDown);
```

//задаются параметры кнопки but

```
this->but->Location=System::Drawing::Point(18,17);  
this->but->Name = L"button1";  
this->but->Size=System::Drawing::Size(95, 99);  
this->but->Text = L"динамическая кнопка";  
this->but->ForeColor=Color::DarkGreen;  
this->but->BackColor=Color::Yellow;
```

```
//означает, что у кнопки есть событие "нажатие на кнопку"
this->but->Click += gnew
System::EventHandler(this, &Form1::but_Click);
this->Controls->Add(this->but);
}
```

Остальной текст пишется после закрывающей скобки обработчика события **Shown** формы **Form1**

```
//определение события кнопки
private: System::Void but_Click(System::Object^
sender, System::EventArgs^ e)
{
//спрятать картинку
dinpib->Hide();}
```

```
//определение события изображения
private: System::Void
dinpib_MouseDown(System::Object^ sender,
System::Windows::Forms::MouseEventArgs^ e) {
```

```
//задаются параметры надписи dinlab
this->dinlab->AutoSize=true;
this->dinlab->Location=
System::Drawing::Point(200,272);
this->dinlab->Name = L"dinlab";
this->dinlab->Size = System::Drawing::Size(41,22);
this->dinlab->ForeColor=Color::Blue;
this->dinlab->Text = L"ЭТО ДИНАМИЧЕСКАЯ КАРТИНКА";
this->Controls->Add(this->dinlab); }
```

```
private: System::Void Form1_Click(System::Object^
sender, System::EventArgs^ e) {
//показать картинку
dinpib->Show();
}
};}
```

Глава 12

Приложения для работы с графикой

12.1. Построение графиков

Рисование объектов происходит в области окна, определенной для этого заранее с помощью метода **CreateGraphics()**. В качестве такой области может быть выбрана как форма, так и другой компонент, например, **PictureBox**.

```
//ссылка на форму, как область для рисования
Graphics ^gr=this->CreateGraphics();
//ссылка на PictureBox, как область для рисования
Graphics ^gr=pictureBox1->CreateGraphics();
```

Теперь все рисуемые графические объекты будут располагаться внутри формы или **PictureBox**, причем их координаты определяются относительно левого верхнего угла выбранной области рисования: по горизонтали – слева направо, по вертикали – сверху вниз. Это необходимо учитывать при задании координат точек, определяемых их местонахождением на плоскости.

Кроме области для рисования нужно определить каким цветом и чем рисовать, то есть создать объект класса **Color**, для последующего определения цвета рисуемого объекта:

```
Color ^zvet= gcnew Color();
```

Затем определить «перо» для рисования данным цветом:

```
Pen ^pero=gcnew Pen(zvet->Red);
```

Или выполнить оба действия одновременно:

```
Pen ^pero =gcnew Pen(Color::Red);
```

Выбирая цвет, удобно использовать суфлер кода.

С помощью конструктора класса **Pen** можно задать также не только цвет, но и ширину пера, а также вид заливки рисуемой фигуры.

Для рисования графических объектов используются методы класса **Graphics**. Рассмотрим некоторые из них:

– **DrawLine()** – линия, соединяющая две точки, каждая из которых определена парой координат:

//линия рисуемая в области, определенной **gr**, пером с параметрами
//аргумента **pero**, координатами первой точки **xt1,yt1** и координатами
//второй точки **xt2,yt2**

```
gr->DrawLine(pero, xt1, yt1, xt2, yt2);
```

– **Clear()** – очистка области рисования и заливка ее определенным цветом.

//область рисования будет белой

```
gr->Clear(Color::White);
```

– **DrawEllipse()** – эллипс, вписанный в прямоугольник.

– **DrawImage()** – изображение в определенном месте.

– **DrawRectangle()** – прямоугольник.

– **FillEllipse()** – внутренняя часть эллипса (заливка).

– **FillRectangle()** – внутренняя часть прямоугольника.

Задание для выполнения 1.

Разработать приложение, определяющее таблицу значений функции e^x для x , изменяющегося от 0 до 11 с шагом 1 и отображающее график этой функции.

Рекомендации для выполнения задания:

– Количество строк и столбцов вводится в соответствующие поля. Количество столбцов вводить всегда равное 2.

– Значения функции рассчитываются по нажатию на кнопку «Результат» и отображаются в компоненте **dataGridView1**.

– График отображается по нажатию на кнопку «График функции».

– При отображении графика не отображается таблица, при отображении таблицы не виден график.

– Использовать множители для того, чтобы весь график был виден на форме.

– Количество строк и столбцов отобразить как глобальные переменные, тогда их значения можно изменять из любого обработчика

событий: `int sk, sv;`

Примечание. График состоит из множества линий, конец каждой из которой определяет начало следующей. Чем меньше будет выбран шаг, тем более точным получится график.

Рекомендуемый интерфейс приложения представлен на рисунке.

The screenshot shows a window titled "Form1" with a table and two input fields. The table has 12 rows and 2 columns. The first column contains indices from 0 to 11, and the second column contains corresponding values. The last row (index 11) is highlighted with a mouse cursor. To the right of the table are two input fields: "Количество столбцов" (Columns count) with the value 2, and "Количество строк" (Rows count) with the value 12. At the bottom of the window are two buttons: "Результат" (Result) and "График функции" (Function graph).

0	1
1	2
2	7
3	20
4	54
5	148
6	403
7	1096
8	2980
9	8103
10	22026
11	59874

The screenshot shows the same window "Form1" but with a graph of the function plotted on the left side. The graph is a smooth curve that starts at the origin (0,0) and increases rapidly, reaching a value of approximately 60,000 at x=11. The input fields and buttons remain the same as in the previous screenshot.

Обработчик события **TextChanged** компонента **textBox1**, используемого для ввода количества строк:

```
//Количество строк компонента dataGridView1 изменяется
//при изменении введенного в поле числа:
dataGridView1->RowCount=Convert::ToInt32
(textBox1->Text);
sk=Convert::ToInt32(textBox1->Text);
```

Обработчик события **TextChanged** компонента **textBox2**, используемого для ввода количества столбцов:

```
//Количество столбцов компонента dataGridView1 изменяется
//при изменении введенного в поле числа:
dataGridView1->ColumnCount=Convert::ToInt32
(textBox2->Text);
sv=Convert::ToInt32(textBox2->Text);
```

Обработчик события «Нажатие на кнопку **Результат**»:

```
//отображение компонента dataGridView1
dataGridView1->Visible=true;

// sk может измениться при добавлении строк в таблицу режиме
// исполнения, определяется количество строк
//компонента dataGridView1
sk=dataGridView1->RowCount;
for(int ig=0;ig<sk;ig++)
{
//заполнение первого столбца таблицы
dataGridView1->Rows[ig]->Cells[0]->Value=
ig.ToString();
int a=Convert::ToInt32(dataGridView1->Rows[ig]->
Cells[0]->Value);

//расчет значений функции и отображение в таблице
```



```

dataGridView1->Rows[ig]->Cells[1]->Value=
Math::Floor( (Math::Exp(a)) ).ToString();
}

```

Обработчик события «Нажатие на кнопку **График функции**»:

```

//компонент dataGridView1 невидим
dataGridView1->Visible=false;

//ссылка на форму, как область для рисования
Graphics ^gr=this->CreateGraphics();

// форма очищается и заливается белым цветом
gr->Clear(Color::White);

//создание пера синего цвета
Pen ^pero=gcnew Pen(Color::Blue);

//объявление координат точек начала и конца линии
int xt1,yt1,xt2,yt2;

//координаты первой точки считываются из первой строки компонента
//dataGridView1
xt1=Convert::ToInt32(dataGridView1->Rows[0]->
Cells[0]->Value);
yt1=Convert::ToInt32(dataGridView1->Rows[0]->
Cells[1]->Value);

//Определение точек для рисования линии, начиная
//со второй строки таблицы
for(int ig=1;ig<sk;ig++)
{
//умножение x*20 и деление y/150 для того,
//чтобы график поместился на форме
xt2=Convert::ToInt32(dataGridView1->Rows[ig]->
Cells[0]->Value)*20;

```

```

yt2=Convert::ToInt32(dataGridView1->Rows[ig]->
Cells[1]->Value)/150;

//вывод отрезка графика функции, как линии
//между двумя точками (xt1,yt1) и (xt2,yt2)

gr->DrawLine(pero, xt1, yt1, xt2, yt2);

//определение точки начала следующего отрезка
xt1=xt2; yt1=yt2;
}

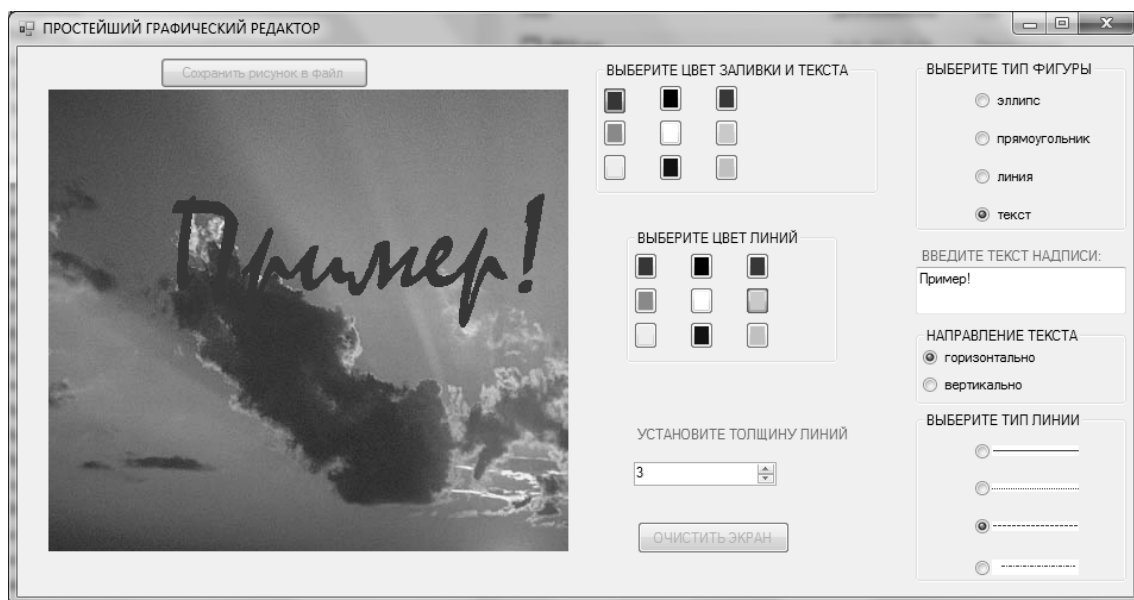
```

12.2. Программирование простейшего графического редактора

Задание для выполнения 2.

Создать приложение «Простейший графический редактор».

Рекомендуемый интерфейс приложения представлен на рисунке.



Требования к приложению:

– при запуске приложения отображается компонент **pictureBox**, в котором отображается изображение по умолчанию;

– координаты точек объявляются как глобальные переменные,

```
int xt1, xt2, yt1, yt2;
```

координаты левой верхней точки изображения любого объекта задаются при нажатии кнопки мыши, при отпускании кнопки мыши задаются координаты правой нижней точки изображения и создается само изображение в зависимости от выбранных параметров;

– для отображения текста текст надписи необходимо ввести в текстовое поле, определить цвет и направление текста, затем выбрать тип фигуры «текст» с помощью компонента **radioButton** с одноименной подписью;

– для выбора цвета заливки и текста, цвета линий используются компоненты **groupBox** и **radioButton** в виде кнопок (установить свойство **Appearance** компонента **radioButton** в значение **Button** и выбрать соответствующее нужному цвету значение свойства **BackColor**);

– для установки толщины линий использовать компонент **numericUpDown** и установить значение свойства **Minimum** равным 1;

– для выбора типа линии использовать компонент **radioButton** и установить его свойство **Image** (рисунки для изображения линий созданы с помощью любого графического редактора);

– обеспечить возможность сохранения полученного изображения в файл и очистки экрана.

Можно самостоятельно усовершенствовать приложение, добавив следующие функции:

– возможность изменения вида шрифта текста, отображаемого на **pictureBox**,

– обработку исключительной ситуации, когда размер шрифта вычисляется неверно (≤ 0);

– путь и имя сохраняемого файла должны указываться с помо-

щью компонента диалога.

– кнопки увеличения и уменьшения размера изображения, главное и контекстное меню, панели инструментов, дополнительные функции графического редактор (например, использовать штриховку).

Обработчик события **MouseDown** компонента **pictureBox** (**нажатие кнопки мыши**):

```
xt1=e->X;  
yt1=e->Y;
```

Обработчик события **MouseUp** компонента **pictureBox** (**отпускание нажатой кнопки мыши**):

```
//необходимо учитывать, что номера компонентов в разрабатываемых  
//приложениях не будут совпадать с предложенным вариантом
```

```
xt2=e->X;  
yt2=e->Y;
```

```
//определяется поверхность рисования
```

```
Graphics ^gr=pictureBox1->CreateGraphics();
```

```
//определяется перо
```

```
Pen ^pero;
```

```
//определяется ширина линии
```

```
int Np=Convert::ToInt32(numericUpDown1->Text);
```

```
//задаются параметры пера - цвет и ширина линии по умолчанию
```

```
pero=gcnew Pen(Color::Red,Np);
```

```
//цвет пера опреляется значением фона компонента radioButton,
```

```
//ширина линии - значением numericUpDown1->Text
```

```
if (radioButton10->Checked==true)
```

```
pero=gcnew Pen(radioButton10->BackColor,Np);
```

```
if (radioButton11->Checked==true)
```

```
pero=gcnew Pen(radioButton11->BackColor,Np);
```

```

if (radioButton12->Checked==true)
pero=gcnew Pen(radioButton12->BackColor,Np);
if (radioButton13->Checked==true)
pero=gcnew Pen(radioButton13->BackColor,Np);
if (radioButton14->Checked==true)
pero=gcnew Pen(radioButton14->BackColor,Np);
if (radioButton15->Checked==true)
pero=gcnew Pen(radioButton15->BackColor,Np);
if (radioButton16->Checked==true)
pero=gcnew Pen(radioButton16->BackColor,Np);
if (radioButton17->Checked==true)
pero=gcnew Pen(radioButton17->BackColor,Np);
if (radioButton18->Checked==true)
pero=gcnew Pen(radioButton18->BackColor,Np);

```

*//стиль пунктирных линий устанавливается в зависимости от компонента **radioButton***

```

if (radioButton26->Checked==true) pero->DashStyle
= System::Drawing::Drawing2D::DashStyle::Solid;
if (radioButton21->Checked==true) pero->DashStyle
= System::Drawing::Drawing2D::DashStyle::Dot;
if (radioButton20->Checked==true) pero->DashStyle
= System::Drawing::Drawing2D::DashStyle::Dash;
if (radioButton19->Checked==true) pero->DashStyle
= System::Drawing::Drawing2D::DashStyle::DashDot;

```

//цвет заливки определяется значением фона компонента radioButton
System::Drawing::SolidBrush^ myBrush; //кисть

//определяются параметры кисти (заливки)

```

if (radioButton1->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush(radioButton1->
BackColor);
if (radioButton2->Checked==true) myBrush=gcnew

```

```

System::Drawing::SolidBrush (radioButton2->
BackColor);
if (radioButton3->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton3->
BackColor);
if (radioButton4->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton4->
BackColor);
if (radioButton5->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton5->
BackColor);
if (radioButton6->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton6->
BackColor);
if (radioButton7->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton7->
BackColor);
if (radioButton8->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton8->
BackColor);
if (radioButton9->Checked==true) myBrush=gcnew
System::Drawing::SolidBrush (radioButton9->
BackColor);

```

//рисуются фигуры

//заполненный эллипс с контуром

```

if (radioButton24->Checked==true)
{gr-> DrawEllipse (pero,xt1,yt1,xt2-xt1,yt2-yt1);
gr->FillEllipse (myBrush,xt1,yt1,xt2-xt1,yt2-yt1);}

```

//заполненный прямоугольник с контуром

```

if (radioButton23->Checked==true)
{
gr->FillRectangle (myBrush,xt1,yt1,xt2-xt1,yt2-
yt1);
}

```

```

gr->DrawRectangle (pero, xt1, yt1, xt2-xt1, yt2-yt1);
}

//линия
if (radioButton22->Checked==true)
gr->DrawLine (pero, xt1, yt1, xt2, yt2);

//текст
if (radioButton25->Checked==true)
{
//высота символов
int M=yt2-yt1;
//текст, выводимый на экран из поля
String^ drawString = textBox1->Text;

//задается формат шрифта и высота символов
System::Drawing::Font^ drawFont=
gcnew System::Drawing::Font("Mistral", M);

//задаются сведения о структуре текста
System::Drawing::StringFormat^ drawFormat =
gcnew System::Drawing::StringFormat();

//направление текста устанавливается вертикальным
if (radioButton29->Checked==true)
drawFormat->FormatFlags=
StringFormatFlags::DirectionVertical;

//создает указываемую текстовую строку в заданном месте - xt1 и yt1
и с указанными параметрами шрифта, текста и цвета
gr->DrawString(drawString, drawFont, myBrush,
xt1, yt1, drawFormat);
}

```

Обработчик события «Нажатие на кнопку **Сохранить рисунок в файл**»:

```
//определяет размер и координаты области копируемой в файл
Rectangle ^r = pictureBox1->
RectangleToScreen (pictureBox1->ClientRectangle) ;

//определяет копируемый рисунок
Bitmap ^b = gcnew Bitmap (r->Width, r->Height) ;

//создает объект из указанного рисунка
Graphics^ g=Graphics::FromImage (b) ;

//передает данные о цвете точек изображения
g->CopyFromScreen (r->Location, Point (0, 0) , r->Size) ;
//сохраняет рисунок в файл с именем 111.jpg на диске d
b->Save ("d:\\111.jpg") ;
```

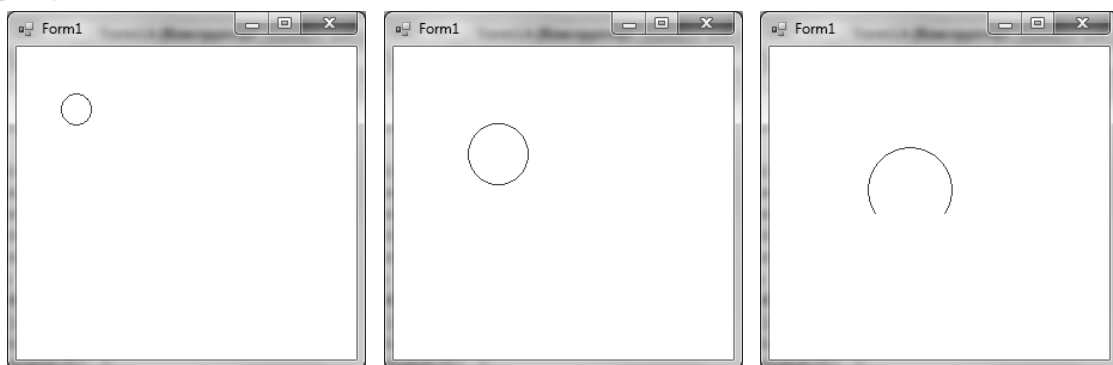
Обработчик события «Нажатие на кнопку **Очистить экран**»:

```
Graphics ^grm=pictureBox1->CreateGraphics() ;
grm->Clear (Color::White) ;
```

12.3. Программирование «движения» графического объекта **Задание для выполнения 3.**

Разработать приложение, в котором графический объект перемещается, постепенно приближаясь и исчезая из поля зрения.

Рекомендуемый интерфейс работающего приложения, зафиксированные через определенные промежутки времени, представлен на рисунке.



Рекомендации для выполнения задания:

– В качестве перемещаемого объекта использовать окружность (эллипс, вписанный в квадрат).

– Область рисования – компонент **Panel**, цвет фона (**BackColor**) у **Panel** и у формы установить белый.

– Перемещение объекта происходит за счет его поочередной прорисовки синим (для отображения на экране) и белым (чтобы удалить предыдущее положение объекта) цветом. Для выполнения данных действий в добавленном в приложение заголовочном файле **A.h** определить класс **A**, в котором содержатся:

закрытые данные целого типа:

//координаты левого верхнего угла рисуемого элемента

int xa, ya,

//величина приращения для определения высоты и ширины

ra;

открытые функции:

конструктор класса;

функция **void shg(Panel ^canvas)** для прорисовки элемента;

функция **void hide(Panel ^canvas)** для «стирания» элемента.

Файл **a.h**

```
#pragma once
```

```
using namespace System::Windows::Forms;
```

```
using namespace System::Drawing;
```

```
ref class A
```

```
{int xa, ya, ra;
```

```
public:
```

```
//конструктор
```

```
A(int xa1,int ya1,int ra1)
```

```
{xa=xa1;ya=ya1;ra=ra1;}
```

```
//функция прорисовки объекта синим цветом
```

```
void shg(Panel ^canvas)
```

```
{
```

```

Graphics ^ g;
g=canvas->CreateGraphics();
Pen ^pero=gcnew Pen(Color::Blue);
g->DrawEllipse(pero,xa,ya,xa+ra,ya+ra);
}

// функция прорисовки объекта белым цветом
void hide(Panel ^canvas)
{
    Graphics ^ g;
    g=canvas->CreateGraphics();
    Pen ^pero =gcnew Pen(Color::White);
    g->DrawEllipse(pero,xa,ya,xa+ra,ya+ra);
}
};

```

– Для обеспечения доступа к функциям класса из обработчиков событий необходимо подключить в файл **Form1.h** заголовочный файл класса **A.h**.

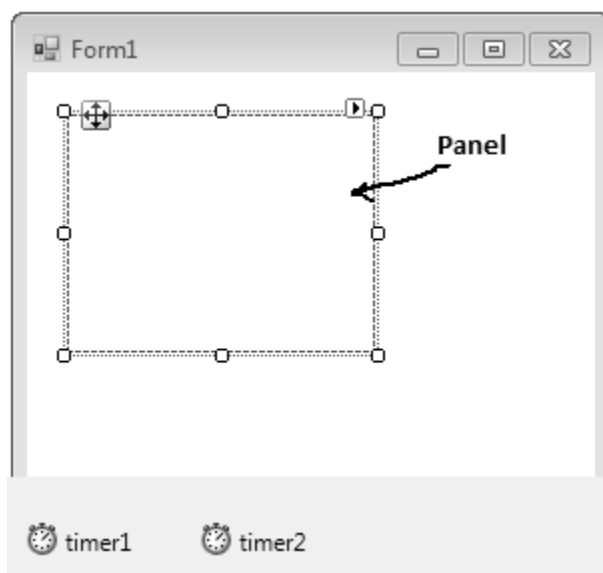
– Объявить как глобальные в файле **Form1.h** (после **using namespace...**)

```
int rd=8,xd=5,yd=6,xv,yv;
```

– Для вызова функций класса использовать два компонента **Timer**, установив их свойства:

- **Enabled** – true;
- **Interval** – 500;

Вид формы в режиме **Конструктора** представлен на рисунке.



Обработчик события **Tick** компонента **timer1**

```
A n(xd,yd,rd);
n.shg(panel1);
xd++;yd++;
```

Обработчик события **Tick** компонента **timer2**

```
xv=xd-1;yv=yd-1;
A n(xv,yv,rd);
n.hide(panel1);
```

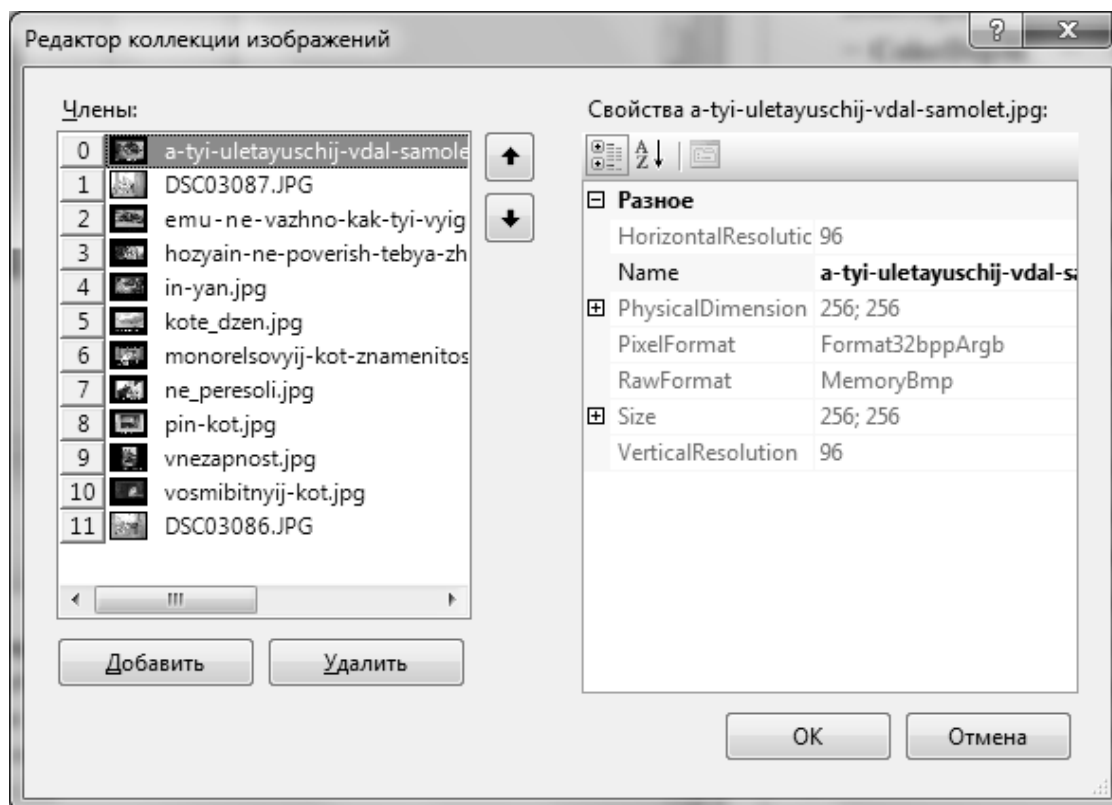
12.4. Особенности работы с компонентом «коллекция рисунков» – **ImageList**

Компонент **ImageList** располагается в окне **Панель элементов** в группе **Компоненты** и используется для хранения группы изображений. Для отображения данных рисунков могут использоваться другие компоненты, у которых имеется свойство **ImageList**. При этом одна и та же коллекция рисунков может использоваться несколькими компонентами.

Некоторые свойства компонента **ImageList**:

- **Images** – хранимые в компоненте изображения. Добавить или удалить изображения из коллекции, а также изменить порядок их отображения можно с помощью окна **Редактор коллекции изображе-**

ний. Данное окно открывается после нажатия кнопки с многоточием в поле данного свойства.



– **ImageSize** – ширина и высота рисунков в коллекции. Рисунки другого размера автоматически адаптируются к заданному здесь размеру.

– **TransparentColor** – определение какого-либо цвета как прозрачного.

– **ColorDepth** – число цветов отображаемых рисунков: **Depth4Bit** – глубина цвета 4 бит, **Depth8Bit** – глубина цвета 8 бит и т.д.

Свойство **ImageList** имеют многие компоненты: **Button**, **CheckBox**, **Label**, **RadioButton**, **TreeView**, **TabControl** и другие, то есть их можно использовать для просмотра коллекции рисунков.

Задание для выполнения 4.

Разработать приложение «Слайд-шоу», где каждую секунду меняется изображение.

Рекомендуемый интерфейс приложения представлен на рисунке.



Рекомендации для выполнения задания:

- для отображения изображений использовать компонент **button1**, с которым связан компонент **imageList1**.
- закрепить **imageList1** за **button1**, выбрав свойство **imageList** – **imageList1**.
- выделить **imageList1**, свойство **Images(Коллекция...)**, нажать кнопку **Добавить...** и выбрать рисунок.
- выбрать первый рисунок, отображаемый на кнопке, для чего в свойстве **ImageIndex** компонента **button1** выбрать из списка рисунков с соответствующим номером.
- установить размер изображения **ImageSize: Width=256; Height=256**
- обеспечить с помощью таймера переключение рисунка через

одну секунду. Для этого установить свойства таймера:

Interval=1000;

Enabled=true;

Обработчик события **Tick** компонента **Timer**

`button1->ImageIndex++;`

– по нажатию на кнопку обеспечить закрытие приложения

Обработчик события **Click** компонента **button1**:

`Close();`

Список литературы

1. Пахомов Б.И. C/C++ и MS Visual C++ 2008 для начинающих. – СПб.: БХВ–Петербург. 2009. – 624 с.
2. Хортон Айвор. Visual C++ 2005: базовый курс.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 1152 с.
3. Холзнер С. Visual C++ 6. Учебный курс. – СПб.: Питер, 2007.
4. Шилдт Г. С++: руководство для начинающих, 2-е издание. : Пер. с англ. – М. : Издательский дом "Вильямс", 2005.
5. Шилдт Г. Самоучитель С++: Пер. с англ. – 3-е изд. – СПб.: БХВ–Петербург, 2005.
6. Гордон Хогенсон. С++/CLI: язык Visual C++ для среды .NET. Пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 464 с.